The Presort Hierarchy for Geometric Problems

Ivor van der Hoog
IT University of Copenhagen, Denmark

Eva Rotenberg
IT University of Copenhagen, Denmark

Jack Spalding-Jamieson
Independent, Vancouver, Canada

Lasse Wulf
IT University of Copenhagen, Denmark

Abstract

Many fundamental problems in computational geometry admit no algorithm running in $o(n \log n)$ time for n planar input points, via classical reductions from sorting. Prominent examples include the computation of convex hulls, quadtrees, onion layer decompositions, Euclidean minimum spanning trees, KD-trees, Voronoi diagrams, and decremental closest-pair.

A classical result shows that, given n points sorted along a single direction, the convex hull can be constructed in linear time. Subsequent works established that for all of the other above problems, this information does not suffice. In 1989, Aggarwal, Guibas, Saxe, and Shor asked: Under which conditions can a Voronoi diagram be computed in $o(n \log n)$ time? Since then, the question of whether sorting along *two* directions enables a $o(n \log n)$-time algorithm for such problems has remained open and has been repeatedly mentioned in the literature.

In this paper, we introduce the Presort Hierarchy: A problem is `1-Presortable` if, given a sorting along one axis, it permits a (possibly randomised) $o(n \log n)$-time algorithm. It is `2-Presortable` if sortings along *both* axes suffice. It is `Presort-Hard` otherwise. Our main result is that quadtrees, and by extension Delaunay triangulations, Voronoi diagrams, and Euclidean minimum spanning trees, are `2-Presortable`: we present an algorithm with expected running time $O(n\sqrt{\log n})$. This addresses the longstanding open problem posed by Aggarwal, Guibas, Saxe, and Shor (albeit randomised). We complement this result by showing that some of the other above geometric problems are also `2-Presortable` or `Presort-Hard`.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Planar geometry, data structures, algebraic decision trees, algorithms with advice

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.75

Related Version *Full Version*: `https://arxiv.org/abs/2602.08843`

Funding This work was supported by the VILLUM Foundation grant (VIL37507) "Efficient Recomputations for Changeful Problems".

Acknowledgements We are thankful to Da Wei Zheng for helpful discussions on extending and simulating Word RAM operations. We are thankful to Jeff Erickson for pointing us to the lower bound of Seidel for the 3D Convex Hull problem.

1 Introduction

Many problems in computational geometry have tight algorithms whose worst-case running time is known to be optimal at $\Theta(n \log n)$. For planar inputs, prominent examples include Pareto fronts, convex hulls, quadtrees, Delaunay triangulations, Voronoi diagrams, well-separated pair decompositions, Euclidean minimum spanning trees, planar triangulations, onion layer decompositions, KD-trees, and decremental closest-pair structures.

The above problems have $\Omega(n \log n)$ *comparison-based* lower bounds: if, for inputs of size n, a problem admits N distinct outputs, then any comparison-based model of computation (such as decision trees or the Real RAM) has an $\Omega(\log N)$ lower bound [8]. A natural question is under which conditions these planar data structures can be constructed in time faster than $O(n \log n)$. Andrew [4] computes a planar convex hull in linear time when the points are given sorted by x-coordinate; the same idea yields a linear-time algorithm for the Pareto front. In contrast, Seidel [37] showed that in $\mathbb{R}^3$, an $\Omega(n \log n)$ lower bound persists even when the input is presorted along any arbitrarily large constant number of directions.

Aggarwal, Guibas, Saxe, and Shor [2] presented in 1989 a linear-time algorithm for constructing the Voronoi diagram of a convex polygon. Since the convexity of the input point set P implies a strong form of presorting – a known radial order along the boundary – they asked which other kinds of presorting information suffice. Chew and Fortune [14] subsequently showed that orthogonal presorting can lead to faster algorithms, but their argument works only for nonstandard Voronoi diagrams. In particular, they give an $O(n \log \log n)$-time algorithm, assuming that the input points are presorted along both the x- and y-coordinates, for constructing Voronoi diagrams under convex distance functions where balls are triangle-shaped. They explicitly ask whether the Euclidean Voronoi diagram can be constructed in $o(n \log n)$ time given sortings along two orthogonal directions. Djidjev and Lingas [17] showed a comparison-based $\Omega(n \log n)$ lower bound for constructing the Euclidean Voronoi diagram even when the input points are sorted by x-coordinate. They also showed a linear-time algorithm for when the input forms a histogram. Buchin and Mulzer [9] mention the existence of an algebraic decision tree of linear depth to construct a quadtree if the point set is presorted along both the x- and y-coordinates. This demonstrates that the standard comparison-based arguments do not rule out faster algorithms, and it motivates the following longstanding open question: "Does there exist an $o(n \log n)$-time algorithm to construct the Euclidean Voronoi diagram of a planar point set that is sorted along both the x- and y-coordinates?"

Equivalence. By classical planar duality, a planar Delaunay triangulation can be transformed into a Voronoi diagram, and vice versa, in deterministic linear time. Krznarić and Levcopoulos [28] showed that a Delaunay triangulation can be transformed into a quadtree in linear time. Buchin and Mulzer [9] gave a randomised linear-time algorithm for constructing a Delaunay triangulation from a quadtree. Subsequently, Löffler and Mulzer [30] established deterministic linear-time algorithms for both transformations. Once a Delaunay triangulation is available, the Euclidean minimum spanning tree (EMST) can be computed in linear time, since the EMST is a subgraph of the Delaunay triangulation. Löffler and Mulzer [30] also show the other direction, as they can construct a Delaunay triangulation from an EMST in linear time. Finally, they also show that quadtrees and well-separated pair decompositions (WSPDs) are equivalent in linear time. Taken together, these results imply that quadtrees, Delaunay triangulations, Voronoi diagrams, WSPDs, and EMSTs form a tightly connected family of geometric data structures: a sub-$O(n \log n)$-time algorithm for constructing any one of them immediately yields such an algorithm for all the others. We refer to these structures as *proximity structures*. Under this equivalence, the open question can be phrased as follows: Does there exist an $o(n \log n)$-time algorithm to construct any proximity structure of a planar point set that is sorted along both the x- and y-coordinates?

Contribution and results. We answer the above question in a positive manner when allowing for randomisation. In full generality, we argue for a *hierarchy* of presortability. A problem is called `1-Presortable` if it admits a (randomised) $o(n \log n)$-time algorithm when the

input is sorted by x. A problem is `2-Presortable` if it is not `1-Presortable` and admits a (possibly randomised) $o(n \log n)$-time algorithm when the input is given sorted along both the x- and y-directions, together with the permutation relating these orderings. Finally, a problem is `Presort-Hard` if it is not `2-Presortable`. We note that the lower bound of Seidel [37] applies even when the input is sorted along any constant number of directions. The same is true for the lower bounds proved in the full version of this paper; however, for ease of exposition, we adopt the above, less general definition of `Presort-Hard`.

Our main contribution is to show that constructing a quadtree is `2-Presortable` on the Real RAM, if randomisation is allowed. Specifically, we present an algorithm with expected running time $O(n\sqrt{\log n})$ (Theorem 5). Combined with the known linear-time reductions between proximity structures, this result resolves the open problems posed by Aggarwal, Guibas, Saxe, and Shor [2] and by Djidjev and Lingas [17], when randomisation is permitted.

We complement this positive result by placing several related geometric problems into this hierarchy (see Table 1). We show that our result implies that finding the maximum empty circle is `2-Presortable`. We also show that KD-trees are `2-Presortable`. Finally, we show in the full version that orthogonal segment intersection is in this class. We prove in the full version that computing the onion layer decomposition is `Presort-Hard`, which in turn implies that decremental convex hulls are `Presort-Hard`. As a consequence, onion layer decompositions do not belong to the family of proximity structures. Finally, we show that ordered k-closest pairs (and therefore decremental closest pairs) are `Presort-Hard`.

Table 1 The known elements in our hierarchy. All the results with labelled theorems or corollaries are novel. We also state whether a result is proven directly, or where it is derived from.

Problem name	Proof technique	Reference
`Presort-Hard`		
3D Convex Hull	Direct	[37]
Ordered k-Closest-Pairs	Reduction from Sorting	Full version
Decremental Closest-Pair	Reduction from Ordered k-Closest-Pairs	Full version
Onion Layer Decomposition	Direct	Full version
Decremental Convex Hull	Reduction from Onion Layer Decomp.	Full version
`2-Presortable`		
Quadtree	Direct	Theorem 5
NN-graph	Reduction to quadtrees	Corollary 6, [30]
Delaunay Triangulation	Reduction to NN-graphs	Corollary 6, [30]
Euclidean MST	Reduction to Delaunay Triangulations	Corollary 6, [30]
Voronoi Diagram	Reduction to Delaunay Triangulations	Corollary 6, [30]
Maximum Empty Circle	Reduction to Delaunay Triangulations	Theorem 10
KD-tree	Reduction to quadtrees	Theorem 8
Orthogonal Segment Intersection	Direct	Theorem 9
`1-Presortable`		
2D Convex Hull	Direct	[4] or [24]
2D Pareto Front	Direct	adapting [4], [24]
Diameter of a Point set	Reduction to 2D Convex Hull	[4] or [24]
A triangulation	Direct	Theorem 7

1.1 Further related work

We briefly mention additional lines of related work. Recently, Eppstein, Goodrich, Illickan, and To [18] presented algorithms for computing the convex hull whose running time depends on the degree of presortedness of the input. Van der Hoog, Rotenberg, and Rutschmann [44] showed that these bounds are tight by establishing matching lower bounds based on an

entropy measure of the partial order. Presorting can also be viewed as a form of *advice* about the input. One may distinguish between *predictions*, which are approximate hints about the output, and *advice*, which is guaranteed to be correct. There is a long tradition on geometric algorithms with predictions [34] and we highlight the recent work of Cabello, Chan, and Giannopoulos [10], who construct a Delaunay triangulation given a predicted triangulation that is close to the true one in terms of flip distance. Regarding *advice*, there are relatively few geometric examples. One notable exception is the framework of *imprecise geometry*, introduced by Held and Mitchell [26], in which each input point p_i is known only to lie within a region R_i. Such regions may be viewed as advice that constrains the space of feasible inputs. Within this framework, efficient algorithms are known for many of the proximity structures discussed above, including convex hulls, Delaunay triangulations, and related constructions [15, 16, 20, 21, 26, 33, 31, 32, 47, 40, 41]. Finally, since many geometric lower bounds are obtained via reductions from sorting, it is worth discussing sorting itself. Sorting with predictions – such as long approximately sorted subsequences – has been investigated in several models [6, 19, 5], while sorting with advice in the form of partial orders or guaranteed comparisons has also received considerable attention [11, 25, 45, 43].

2 Preliminaries

In this paper, the algorithmic input consists of both real-valued and integer data. In full generality, the real-valued input is a sequence $A = (a_1, \ldots, a_r)$ of length r, and the integer input is a sequence $B = (b_1, \ldots, b_m)$ of length m.

Presorting. In this paper, the integer input encodes a *promise* about the real-valued data. One such promise is what we call *presortedness*. We say that the input (A, B) is a 1-*presorting* of a point set P if A stores P sorted along a direction in $\mathbb{R}^2$. We say that the input (A, B) is a 2-*presorting* of a point set P if A stores P twice (sorted along orthogonal directions) and B specifies the permutation between the two sortings. Formally, we say that $r = 2 \cdot n$ and that the real-valued sequence A can be partitioned as $A = A_x \cup A_y$ where A_x sorts P by x and A_y sorts P by y. For notational convenience, we index the points of P so that $A_x[i] = p_i$ for all $i \in [n]$. We assume that the integer input B then specifies the permutation from A_x to A_y. Formally, it is an integer sequence π such that $p_i = A_y[\pi(i)]$ for all $i \in [n]$. We study for which geometric problems it is useful to have access to a presorting.

The Real RAM. We provide both upper and lower bounds for geometric problems whose input contains both real-valued and integer data. It is therefore necessary to define our computational models with some care. Our primary model for upper bounds is the standard model for real-valued geometric computation, namely the *Real RAM*. We adopt the textbook definition of the Real RAM due to Preparata and Shamos [35]. This classical model allows both real-valued and integer operations: it is a Random Access Machine (RAM) in which each memory cell may store either a real number or an integer. The input $x \in \mathbb{R}^r \times \mathbb{N}^m$ occupies the first $r + m$ memory cells, in order. A program operating on input x is a finite sequence of instructions, each of which is of one of the following three types:

- *Primitives* manipulate storage cells, after which execution proceeds to the next instruction.
- *Control-flow* specifies an integer i and reads a memory cell containing a Boolean; if `True`, execution jumps to the ith instruction, and otherwise proceeds to the next instruction.
- *Halts* terminate the program, and the memory at that time constitutes the output.

A *Primitive* takes two memory cells as input and writes its result to a new memory cell. The model supports the following constant-time Primitives:

- Basic arithmetic operations (addition $+$, subtraction $-$, multiplication $\times$, and division $/$).
- Comparison operations (equality $=$ and order $<$).
- Indirect addressing, but only via memory cells that are designated to contain integers.

These Primitives are subject to the following restrictions. Real values cannot be written to integer memory cells. Any arithmetic operation involving at least one real-valued cell must write its result to a real-valued cell; in particular, such results cannot be used for memory addressing. Comparison operations output a Boolean, which is stored as an integer and may be used by Control-flow instructions for branching. Integer division $x, y \mapsto \left\lfloor \frac{x}{y} \right\rfloor$ is *not* a Primitive, even when both inputs are integers. Finally, we allow randomisation, which can be modelled as a Control-flow branch that is taken with probability $\frac{1}{2}$.

Algebraic Decision Trees. To obtain Real RAM lower bounds, we typically apply a Ben-Or [8] argument: counting the leaves in a real-valued *Algebraic Decision Tree* (ADT).

For a *real-valued* ADT with input-size N, the input is a vector $X = (x_1, \ldots, x_N)$ of N real numbers. It is a rooted tree in which each internal node is labelled with a multivariate polynomial $p(x_1, \ldots, x_N) \in \mathbb{R}[x_1, \ldots, x_N]$. Each internal node has three outgoing edges labelled $p(x) < 0$, $p(x) = 0$, and $p(x) > 0$. Each leaf of the ADT contains a *combinatorial output*, that is, a list of integers, where each integer refers to an index of an input variable. For example, when sorting the N numbers, each leaf contains a permutation of $[N]$, where a permutation such as $(1, 3, 7, \ldots)$ indicates that x_1 is the smallest input, followed by x_3, then x_7, and so on. An ADT is *complete* if every fixed input X traverses a path to a leaf. It is *correct* if it is complete and every input leads to the corresponding desired output.

If the input to a Real RAM program is purely real-valued, then real-valued ADTs are strictly more powerful than the Real RAM. This follows from a classical argument where any real-valued Real RAM program constructs an ADT whose depth is at most its worst-case running time [8]. Therefore, any lower bound for real-valued ADTs also applies to the Real RAM when the input is purely real-valued. Since each internal node has three outgoing edges, an ADT with k leaves must have depth at least $\Omega(\log_3 k)$ on some input. Thus, if over all real-valued inputs $X \in \mathbb{R}^N$ there are k distinct combinatorial outputs, then both real-valued ADTs and the Real RAM admit a lower bound of $\Omega(\log_3 k)$. This also holds in expectation, since the median depth of the tree is $\Omega(\log_3 k)$.

Integer inputs and ADTs. In many cases, however, the algorithmic input also contains integers (see, for example, the lower bounds by Seidel [37]). If the input contains integers, the computational strength of a real-valued ADT and the Real RAM are incomparable: as the Real RAM can use the integer input for indirect addressing and outputting a solution. We give a concrete example: let the real input $A \subset \mathbb{R}^n$ specify a real-valued point set P and the integer input $\pi \subset \mathbb{N}^n$ specify the sorting of P along a space-filling curve (for each index i, the element $A[\pi(i)]$ is the i'th element along the sorting order). If the goal is to sort P along this space-filling curve, then a Real RAM program can simply output π. Yet, any program that can only receive real-valued input is required to sort A.

Clairvoyant ADTs. As a consequence, lower bounds for real-valued ADTs are not lower bounds for the Real RAM when presortings or other integer-valued promises are provided, and we have to adopt a different strategy. Let the input be a real-valued vector A and some integer vector B. For any fixed B, we define a *clairvoyant* ADT $\mathcal{A}_B$ in a similar way

to [1, 42]: It is a real-valued ADT that knows B as a constant and receives A as input. For any Real RAM program with input $A \cup B$, a clairvoyant ADT $\mathcal{A}_B$ is strictly more powerful. In particular, consider any lower bound L on the maximum depth traversed by a clairvoyant ADT $\mathcal{A}_B$ over all real-valued inputs A such that $A \cup B$ is a valid input. Then L is also a worst-case lower bound for any Real RAM program. Clairvoyant ADTs are thus a powerful tool for lower bounds. For upper bounds they are less applicable because they are computationally quite strong: any Real RAM or ADT cannot simulate a clairvoyant ADT $\mathcal{A}_B$ since it would have to identify the input B and "find" the corresponding program or tree.

Word RAM. Finally, we mention the Word RAM model as given by Fredman [23]. The reason we mention this model is that we will be simulating its instructions on a Real RAM (under some assumptions about what those operations operate on). Let the input size be N. In the Word RAM model, we have a machine with an unbounded sequence of memory cells, each storing an integer of $O(\log N)$ bits. A program is again a finite sequence of instructions, of *Primitives*, *Control-flow* and *Halts*. A *Primitive* takes k memory cells as input and writes its result to a memory cell. The model supports the following constant-time Primitives:

- Integer arithmetic (addition $+$, subtraction $-$, multiplication $\times$, and integer division).
- Comparison operations (equality $=$ and order $<$).
- Direct and indirect memory addressing.

Each Primitive operating on k memory cells takes k time. In our use of this model, we will only store integers in $[2n]$, and k will always be constant. As we will discuss in Section 5, many other word-level operations such as finding the most significant bit can, after linear-time preprocessing, be simulated in constant time [23].

3 Quadtrees via Linear-depth Clairvoyant Algebraic Decision Trees

Buchin and Mulzer [9] mention the existence of an algebraic decision tree of linear depth to construct a quadtree of a presorted point set (A_x, A_y, π), although they do not provide details. Via private communication, we confirmed that Buchin and Mulzer [9] envisioned a specific construction of a linear-depth clairvoyant ADT. Details of their construction will be helpful for our algorithm in Section 4. Hence, in this section, we present such a construction:

▶ **Theorem 1** (Buchin and Mulzer [9]). *Let $\pi\colon [n] \to [n]$ be a permutation. Denote by $\mathbb{I}_\pi$ all presorted point sets $(A_x \cup A_y) \times \pi$ where $\forall i \in [n]$, $A_x[i] = A_y[\pi(i)]$. There exists a clairvoyant decision tree $\mathcal{T}_\pi$ of linear depth that for all $(A_x, A_y) \in \mathbb{I}_\pi$ finds the corresponding quadtree.*

Open and closed squares. A quadtree is, on a high level, a partitioning of a bounding square B into smaller squares such that each square contains at most one point of the input P. To formally define such a partition, we need to distinguish between open and closed squares. A closed square is the area $[a, a+c] \times [b, b+c]$ for positive integers a, b and c. With abuse of terminology, we define the open square as the area $[a, a+c) \times [b, b+c)$ and we say that the squares $[a, a+c) \times [b, b+c]$ and $[a, a+c] \times [b, b+c)$ are *half-open*. In the definition of a (compressed) quadtree, we must carefully partition the bounding square B into closed, open, and half-open squares such that each point of P lies in exactly one square.

Quadtrees. Let P be a planar point set contained in some (open, half-open, or closed) axis-aligned square B. The *split* operation $\texttt{split}(B)$ is the unique partitioning of B into four equal-area closed, half-open and open squares such that each point $q \in B$ is contained in a unique part. A *quadtree* is a tree obtained by recursively applying the split operation on

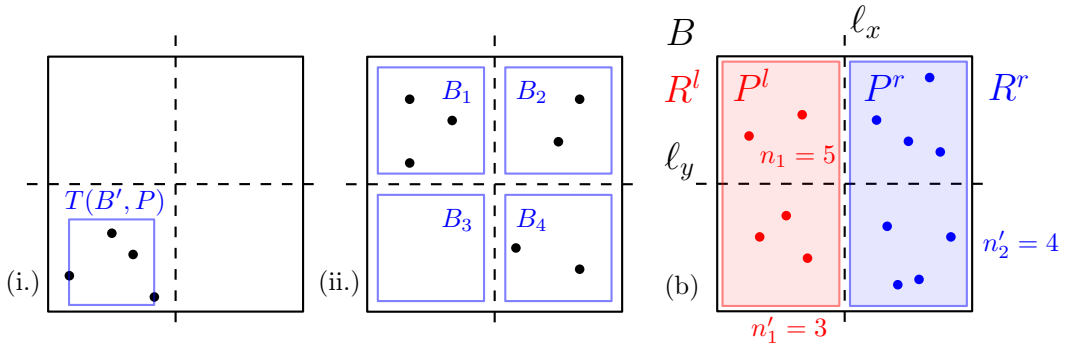

Figure 1 Depiction of type i. and type ii. quadtree splits. (b) Our variable names.

squares C in B where $P \cap C$ contains at least two points. In the resulting tree T, we do not distinguish between nodes v of T and their corresponding square in the plane. The depth of the resulting tree is in $\Omega(\log \Psi)$, where Ψ is the ratio between the distance realised by the farthest pair of points and the closest pair of points. Since P may be an arbitrary size-n planar point set, this quantity is not necessarily bounded.

Compressed quadtrees. To avoid having unbounded-depth trees, we typically define and compute a *compressed quadtree* [30]. Morally, a compressed quadtree considers maximal descending paths $v_0 \to v_1 \to \ldots \to v_\ell$ in the quadtree such that $v_1 \cap P = v_\ell \cap P$ and, if this path has length 2 or greater, makes v_0 the parent of v_ℓ. Formally, we are unable to compute this quadtree cell v_ℓ on a Real RAM as it effectively corresponds to computing the maximum integer k such that there exist integers (i, j) such that (a subset of) P is contained in the square $[i \cdot 2^{-k}, (i+1) \cdot 2^{-k}) \times [j \cdot 2^{-k}, (j+1) \cdot 2^{-k})$. This in turn requires integer arithmetic operations that are not available on the Real RAM. Instead, compressed quadtrees are recursively defined:

▶ **Definition 2** (Figure 1). *Let B be an (open, half-open, or closed) axis-aligned bounding box containing a planar point set P. Let $(B_1, B_2, B_3, B_4) = \texttt{split}(B)$. The* compressed *quadtree $T(B, P)$ has as its root B which is split according to one of two types:*

- **i.** *If P is entirely contained in one of the four squares B_i, then let $B' \subset B_i$ be a minimum-area closed square containing P. The root B has a single child which is $T(B', P)$.*
- **ii.** *Otherwise, B has four children which are $T(B_i, P \cap B_i)$ for $i \in [4]$.*

3.1 A Linear-depth Clairvoyant Algebraic Decision Tree

Constructing a compressed quadtree reduces to sorting, and therefore there cannot exist a Real RAM algorithm whose (expected) running time is $o(n \log n)$ without additional information. Yet (Theorem 1), for a fixed permutation π there exists a clairvoyant ADT $\mathcal{T}_\pi$ of linear depth such that for all 2-presortings $(A_x \cup A_y) \times \pi$, the ADT finds the quadtree of P. This is *not* sufficient to create an algorithm which constructs the quadtree for an arbitrary presorted point set – this is because $\mathcal{T}_\pi$ can be wildly different for different π, and it is a priori not clear how to efficiently construct $\mathcal{T}_\pi$ given π. Nonetheless, the idea behind the linear-depth ADT $\mathcal{T}_\pi$ will be the first building block for our more powerful algorithm in the later Section 4. We now explain in detail how to construct such an ADT. We note that the original idea originates from Buchin and Mulzer [9]. Our tree constructs the quadtree top down and recursively. Specifically, we assume as recursive input a tuple (P, n, B, A_x, A_y):

▶ **Invariant 1.** *We maintain the invariant that our recursive input is a point set P of size n, an open, half-open or closed square B, and two input arrays A_x and A_y that contain P sorted by x- and y-coordinate respectively. The algorithm will perform either a Type 1 or Type 2 split on B, and recurse on each resulting square with two corresponding sorted arrays.*

Splitting B. We describe a procedure that determines which of the Type 1 or Type 2 splits the square B requires. If a Type 2 split is required, we create the recursive input according to Invariant 1. The main idea is to use a two-sided exponential search approach in order to make sure that this information is propagated using only sublinear depth in the ADT. Together with the recursive inequality described in Appendix B, this will yield the result. The procedure runs in two stages, first for splitting along x and then along y.

First, consider the following notation, depicted in Figure 1 (b). Let ℓ_x and ℓ_y be the vertical and horizontal lines splitting B in half (geometrically). We denote by R^l and R^r the two rectangles formed by partitioning B along ℓ_x. We define $P^l = P \cap R^l$ and $P^r = P \cap R^r$. Finally, we define the 2-presortings (A^l_x, A^l_y, π^l) and (A^r_x, A^r_y, π^r) of P_l and P_r. Our first stage computes R^l, R^r and these presortings the following way. Let $n_1 = |P^l|$ denote the number of points of P that lie strictly left of ℓ_x (inside R^l). We first consider a branching of our decision tree into three branches: where either $n_1 = 0$, $n_1 = n$, or $n_1 \in [1, n-1]$. Note that we can identify in which of these three cases we are using a subtree of constant height by comparing ℓ_x to $A_x[1]$ and $A_x[n]$. If $n_1 = 0$, we conclude the first phase by reporting R^l with $(A^l_x, A^l_y, \pi^l) = (\texttt{null}, \texttt{null}, \texttt{null})$ and R^r with $(A^r_x, A^r_y, \pi^r) = (A_x, A_y, \pi)$. If $n_1 = n$, we set $(A^l_x, A^l_y, \pi^l) = (A_x, A_y, \pi)$ and R^r with $(A^r_x, A^r_y, \pi^r) = (\texttt{null}, \texttt{null}, \texttt{null})$. In both of these cases, the branches then enter the second phase.

In the third case, the branch where $n_1 \in [1, n-1]$, we place a subtree that simulates exponential search over A_x. This search iteratively compares for an index j the point $A_x[j]$ to the line ℓ_x. This procedure finds the maximum index i such that $A_x[i]$ lies left of ℓ_x using a path of length $O(\log(\min\{i, n-i\}))$. Consider the branch corresponding to some fixed i. In this branch, we can obtain an x-sorted array A^l_x storing P_l by splitting A_x on i. Obtaining A^l_y is more work: for each distinct value i, consider the permutation π restricted to $[1, i]$. This gives a set of indices $I_i \subset [1, n]$ and the array A^l_y equals all elements $A_y[j]$ for $j \in I_i$, where the j's are ordered from low to high. Thus, given i and π, there exists some fixed injective map $\pi_i \colon [i] \to [n]$ such that A^l_y is equal to the ordered sequence $A_y[\pi_i(j)]$ for $j \in [i]$. It follows that the branch $\mathcal{B}_i$ of the decision tree that corresponds to splitting A_x on index i can implicitly consider the map π_i to obtain A^l_y and thus a presorting (A^l_x, A^l_y, π_i) of P_l. By *implicitly*, we mean the following: Since π is fixed, it means that once we are inside the branch $\mathcal{B}_i$, the permutation π_i is unambiguously described, and contains the information about which points constitute the preordering $A^l_y[j]$ for $j \in [i]$. Hence in all future recursive calls, whenever the decision tree would need to reference an element $A^l_y[j]$ of the sorted order A^l_y, it can reference $A_y[\pi_i(j)]$ instead. Note that since we are only interested in the existence of a decision tree, and not how to efficiently construct it, this implicit operation is allowed.

Presorting P_r is done analogously. It follows that in all three cases we enter the second phase having computed R^l and R^r and the presortings (A^l_x, A^l_y, π^l) and (A^r_x, A^r_y, π^r). The path to this branch had length $O(1 + \log(\min\{i, n-i\})) = O(1 + \log(\min\{n_1, n-n_1\}))$.

The second phase. Having halved B by ℓ_x, our decision tree branches into the second phase. Let ℓ_y be the horizontal line that splits B in half. The line ℓ_y splits R^l into two squares B_1 and B_2 where we define $P_1 = P^l \cap B_1$ and $P_2 = P^l \cap B_2$ (see Figure 1 (b)). Let $n'_1 = |P_1|$ denote the number of points that lie strictly below ℓ_y (these lie inside P_1). We use

the exact same technique as above to obtain a presorting (A_x^1, A_y^1) of P_1 (and a presorting (A_x^2, A_y^2) of P_2) after a path of depth $O(1 + \log(\min\{n_1', |P^l| - n_1'\}))$. We define B_3 and B_4 off of R^r, with $P_3 = P^r \cap B_3$ and $P_4 = P^r \cap B_4$ analogously and obtain a presorting of P_3 and P_4 using a path of depth $O(1 + \log(\min\{n_2', |P^r| - n_2'\}))$ where $n_2' := |P_3|$.

Denote $(B_1, B_2, B_3, B_4) = \texttt{split}(B)$, define $P_i := P \cap B_i$. Denote by n_1 the number of points in $B_1 \cup B_2$ and by $n_2 = n - n_1$. Denote by n_1' the number of points in B_1 and by n_2'' the number of points in B_3. The above procedure defines a subtree that computes for each i the point set P_i and a 2-presorting of P_i using a path of depth:

$$O\left(1 + \log(\min\{n_1, n_2\}) + \log(\min\{n_1 - n_1', n_1'\}) + \log(\min\{n_2 - n_2', n_2'\})\right) \tag{1}$$

Determining the split type and tree-depth analysis

Note that given $(B_1, B_2, B_3, B_4) = \texttt{split}(B)$, define $P_i := P \cap B_i$, we can determine the split type of the quadtree. Specifically, we perform a Type 1 split if and only if there exists a unique i such that $P_i = P$. Our ADT defines a minimum-area closed bounding square $B' \subset B_i$ as a constant-complexity expression using $\{A_x[1], A_x[n], A_y[1], A_y[n]\}$. We can then recurse on (P, B', A_x, A_y). Otherwise, we perform a split of Type 2. Per construction, our ADT has obtained a 2-presorting of each point set $P_i = P \cap B_i$ and so this also maintains our input invariant. What remains is to analyse the total depth of this tree.

First, we note that splits of Type 1 increase the depth of our tree by at most a constant. Specifically, Type 1 occurs if and only if $n_1, n_2 \in \{n, 0\}$ and $n_1', n_2' \in \{n, 0\}$ and so identifying a Type 1 split takes constant depth. We cannot compute two Type 1 splits in a row as the new bounding box B' has two points of P on opposite facets. I.e., the horizontal or vertical middle of B' must separate at least two points of P.

To bound the tree depth, we switch our perspective slightly. Let us denote by $T_x(n)$ the remaining depth of the ADT when considering the point set P of size n (before performing the x-split). Let us further denote by $T_y^l(n_l)$ the remaining depth of the ADT when considering the point set P^l of size n_l. Likewise denote by $T_y^r(n_r)$ the remaining depth of the ADT when considering the point set P^r of size n_r. For some $n \in \mathbb{N}$, let $\hat{T}_x(n)$ denote the maximum of $T_x(n)$ over all point sets P of size n that can be encountered throughout the ADT. Likewise, let $\hat{T}_y^l(n)$ ($\hat{T}_y^r(n)$ respectively) denote the maximum of $T_y^l(n)$ over all point sets P^l of size n_l (the maximum of $T_y^r(n)$ over all point sets P^r of size n_r respectively). We have

$$\hat{T}_x(n) \leq \max_{1 \leq n_1 < n} \left(\hat{T}_y^l(n_1) + \hat{T}_y^r(n - n_1) + O(1 + \log(\min(n_1, n - n_1))) \right)$$

we have similar recursions for $\hat{T}_y^l(n_1), \hat{T}_y^r(n)$. We define $T(n) := \max\{\hat{T}_x(n), \hat{T}_y^l(n), \hat{T}_y^r(n)\}$

$$\text{and obtain:} \quad T(n) \leq \max_{1 \leq i < n} T(i) + T(n - i) + O(1 + \log(\min(i, n - i))).$$

This recursive inequality evaluates to $O(n)$. For completeness, we show this argument in Appendix B. Therefore the clairvoyant ADT $\mathcal{T}_\pi$ has linear depth, and we conclude Theorem 1.

4 A randomised $O(n\sqrt{\log n})$ time algorithm for Quadtree construction

Djidjev and Lingas [17] showed that Voronoi diagrams (and thus quadtrees) are not `1-Presortable`. In this section, we show that quadtrees are `2-Presortable` as we present a randomised $O(n\sqrt{\log n})$-time Real RAM algorithm to construct a quadtree from a 2-presorted point set (A_x, A_y, π). One of our key insights is to use the efficient orthogonal

range searching data structure by Belazzougui and Puglisi [7, Theorem 4]. This is a Word RAM algorithm that, after $O(n\sqrt{\log n})$-time preprocessing, can, for any rectangular range, locate the rightmost, leftmost, topmost or bottommost point in the query. We abuse the fact that the permutation π specifies inputs in the range $[n]$ to simulate the required Word RAM instructions at no overhead. To not distract from our main argument, we show in Section 5 the following:

▶ **Theorem 3.** *Let π be a size-n integer array that specifies a permutation from $[n]$ to $[n]$. Let $I_\pi = \{(i, \pi(i)) : i \in [n]\}$ be the induced point set in $[1, n]^2$. On a Real RAM we can, with $O(n\sqrt{\log n})$ preprocessing, store I_π in a linear-size data structure that can answer the following queries in $O(\log^\varepsilon(n))$ time (for any fixed $\varepsilon > 0$):*

- *`xNext(rectangle` $R \subset [1, n]^2$`)` reports the leftmost and rightmost point in $I_\pi \cap R$.*
- *`yNext(rectangle` $R \subset [1, n]^2$`)` reports the topmost and bottommost point in $I_\pi \cap R$.*

Assuming the existence of the auxiliary data structure, we compute from a 2-presorted point set P given as (A_x, A_y, π) its quadtree in $O(n\sqrt{\log n})$ expected time. We denote by $[1, n]^2 \cap \mathbb{N}^2$ the *rank space*. The point set P has a corresponding point set R_P in rank space where $(i, j) \in R_P$ if there exists a point $p \in P$ where $A_x[i] = p$ and $A_y[j] = p$. We denote by $\rho\colon \mathbb{R}^2 \to [1, n]^2 \cap \mathbb{N}^2$ the map that maps any $p \in P$ to its corresponding point in rank space.

Morphing squares into rank-space rectangles. Let $B \subset \mathbb{R}^2$ be any axis-aligned square and let (A_x, A_y, π) be a presorting of a planar point set P. Let p_ℓ, p_r, p_b, and p_t denote the leftmost, rightmost, bottommost, and topmost points of $B \cap P$, respectively. We write $\texttt{xIndex}(p)$ for the index of a point p in A_x, and $\texttt{yIndex}(p)$ for its index in A_y. Observe that every point $p \in P \cap B$ satisfies $\texttt{xIndex}(p) \in [\texttt{xIndex}(p_\ell), \texttt{xIndex}(p_r)]$ and $\texttt{yIndex}(p) \in [\texttt{yIndex}(p_b), \texttt{yIndex}(p_t)]$. Conversely, any point whose indices lie in these two intervals lies in B. We therefore associate with B the axis-aligned rectangle in rank space $\Gamma(B) := [\texttt{xIndex}(p_\ell), \texttt{xIndex}(p_r)] \times [\texttt{yIndex}(p_b), \texttt{yIndex}(p_t)]$, which is the unique minimum-area rectangle with:

$$p \in P \cap B \qquad \text{if and only if} \qquad (\texttt{xIndex}(p), \texttt{yIndex}(p)) \in \Gamma(B)$$

Preprocessing P. Before computing the quadtree of P, we preprocess P in the following manner. We create subsets $P_0 \supset P_1 \supset P_2 \ldots$ via a stochastic process where $P_0 = P$ and P_i is constructed by sampling every point from P_{i-1} at probability $\frac{1}{2}$. We store for each of the expected $O(\log n)$ point sets P_i, its corresponding subset $R_{P_i} \subset R_P$ in the auxiliary data structure $D(R_{P_i})$ from Theorem 3. Since the sizes of P_i follow, in expectation, a geometric sum this takes expected $O(n\sqrt{\log n})$ time.

Our recursive invariant. After preprocessing, we construct the quadtree top-down; splitting a square B and recursing on its children. The recursive invariant used in Section 3.1 is too expensive to maintain on a Real RAM. Rather, we maintain a lighter invariant where we assume that our recursive input is an open, half-open, or closed square B and the leftmost, rightmost, bottommost, and topmost point of $P \cap B$. Note that we can identify that B is a leaf of the quadtree by testing if these four points are the same. From here, we simulate the algorithm from Section 3.1 at $O(\log^\varepsilon n)$ overhead using the queries from Theorem 3.

▶ **Lemma 4.** *Let B be a rectangle and denote by ℓ_x the vertical line splitting B in half. Denote by R^l and R^r the two rectangles formed by partitioning B along ℓ_x and define $P^l = P \cap R^l$ and $P^r = P \cap R^r$. Given B, the leftmost, rightmost, bottommost, and topmost point of $P \cap B$ and our auxiliary data structures, we can determine the leftmost, rightmost, bottommost, and topmost points of P^l and P^r in expected $O((1 + \log(\min\{|P^l|, |P^r|\})) \log^\varepsilon n)$ time.*

Proof. Consider the sequence σ of all points in $P \cap B$, sorted by x-coordinate. Then the leftmost and rightmost points, p_l and p_r in $P \cap B$, are at the start and end of σ. Denote by q the rightmost point in P^l then q is at index $b = |P^l|$ in σ. We wish to find q from p_l in $O(\log b)$ time. We achieve this by performing exponential search as in a skip list:

Consider the point sets $P_0, P_1, \ldots$. The probability that a point $p \in P$ is in P_i equals 2^{-i}. Now consider all points in $\sigma[1, b]$. We define for every such point p its *height* as the highest index i such that $p \in P_i$. We denote by $p_{max} \in \sigma[1, b]$ the node from P^l with the highest height i. In expectation, it holds that $i \in O(\log b)$. This enables the following algorithm:

Overall algorithm. Given the leftmost, rightmost, bottommost, and topmost point of $P \cap B$, we compute $\Gamma(B)$ in constant time. We set $G = \Gamma(B)$ and alter it by moving its left boundary by one. From $i = 0$, we query the level-i data structure $D(R_{P_i})$ with $\texttt{xNext}(G)$ to find the leftmost point in G. If the result lies left of ℓ_x, we increment i, continuing until we find p_{max}. We then shrink G by setting its left boundary to $\texttt{xIndex}(p_{max}) + 1$. We set a counter c to i and decrement it once. We finally apply the following skip-list algorithm (see Figure 2):

- While $\texttt{xNext}(G)$ on $D(R_{P_c})$ finds a point p left of ℓ_x, shrink G by setting its left boundary to $\texttt{xIndex}(p) + 1$. We call this a *right-step*.
- Otherwise, decrement c by one. We call this a *down-step*.
- If we cannot perform a right-step and $i = 0$, we have found q.

This procedure always finds q since the left boundary of G never corresponds to a point right of ℓ_x, and $q \in P_0$. We now argue that the expected running time of this procedure is only $O(\log b)$ plus $O(\log b)$ queries to the data structure. We will use very similar arguments as in the original analysis of the skip list data structure introduced by Pugh [36]. First, observe that the procedure first finds p_{max}. Then the counter c (which can be interpreted as the current height of the search) only ever decreases until q is found. Note that the height i of p_{max} in expectation is $O(\log b)$, but depending on the randomness used could be even larger. Based on this, we divide the procedure into three phases.

1. The phase until p_{max} is found
2. The phase during which the counter c satisfies $c \geq \log b$
3. The remaining phase where $c < \log b$ until q is found.

We limit the expected running time of each phase separately. Phase one has running time $O(i)$ where i is the height of $p_{\max}$, hence $O(\log b)$ in expectation.

The running time of phase two is equal to the number M_1 of times c was decremented during phase two, plus the number M_2 of right steps performed during phase two. We have $M_1 \leq i$, hence $\mathbb{E}[M_1] \leq \log b$. Furthermore, M_2 is bounded by the number of elements in the array $\sigma[1, b]$ whose height is $\log b$ or higher. Hence we have $\mathbb{E}[M_2] \leq b2^{-\log b} = 1$.

Finally, the running time of phase three is analysed using a classical skip-list argument: For each $k = \log b, \log b - 1, \ldots, 1$, let Y_k be the number of steps the algorithm performs during phase three from meeting the first element at height k until meeting the first element at height $k - 1$. Pugh showed that $\mathbb{E}[Y_k] = O(1)$ [36]. Indeed, we can imagine the search occurring backwards, and the randomness that generates the height of the elements being generated on-the-fly. Under this interpretation, if the current element has height $k - 1$, and we consider some element x before the current element with height k', we have a backwards pointer if and only if $k' \geq k - 1$ and we go up a level if and only if $k' \geq k$. Then for each backwards pointer, the probability that we go up a level is at least $1/2$. This implies that $\mathbb{E}[Y_k] = O(1)$. In total, the expected runtime of phase three is $\sum_{k=2}^{\log b} \mathbb{E}[Y_k] = O(\log b)$.

We can perform a symmetric procedure from p_r. Specifically, we can query $\texttt{xNext}(G)$ to find the rightmost point in G and altering the right boundary instead, to find q using in expectation $O(1 + \log(\min\{|P^l|, |P^r|\}))$ queries. Given q, we have found the leftmost and

rightmost points of R^l. Doing a symmetric procedure gives the leftmost and rightmost points in R^r. Finally, given q, we partition $\Gamma(B)$ into two rectangles where the first rectangle R_1 is $\Gamma(B)$ up to $\texttt{xIndex}(q)$ and the second R_2 is $\Gamma(B)$ from $\texttt{xIndex}(q)+1$ rightwards. We query these two rectangles with the `yNext` queries on $A(P_0)$ to find the topmost and bottommost points in R^l and R^r. ◀

▶ **Theorem 5.** *Let P be a size-n point set given as (A_x, A_y, π) where A_x and A_y sort P by x and y respectively, and π is the permutation that maps A_x to A_y. We can construct a quadtree on P in expected $O(n\sqrt{\log n})$ time.*

Proof. Our preprocessing required expected $O(n\sqrt{\log n})$ time. We then split a square B, and recurse on its children maintaining our recursive invariant that for the input B we have the leftmost, rightmost, bottommost, and topmost point of $P \cap B$. The square B is a leaf of the quadtree if and only if these points are all the same (or *null* for an empty square). Such a split is performed in the following manner. Let $(B_1, B_2, B_3, B_4) = \texttt{Split}(B)$ and denote by ℓ_x the vertical line splitting B in two and by ℓ_y the horizontal line. We use Lemma 4 to run the exact algorithm as in Section 3.1:

We split B into R^l and R^r in expected time $O((1+\log(\min\{|P\cap R^l|, |P\cap R^r|\}))\log^\varepsilon n)$. We then split R^l into (B_1, B_2) and R^r into (B_3, B_4) in expected time $O(\log^\varepsilon n \cdot (1+\log(\min\{|P\cap B_1|, |P\cap B_2|\}) + \log(\min\{|P\cap B_3|, |P\cap B_4|\})))$. If there exists an index i such that $B\cap P = B\cap P_i$ we can detect this case because for all other B_j, the topmost point is *null*. We perform a Type 1 split using the leftmost, rightmost, bottommost, and topmost point of $P\cap B$ and we recurse on the new child B'. As in Section 3.1, we are guaranteed that this recursive call generates a Type 2 split next. For Type 2 splits, we recurse on at most four subproblems B_i of size $|P\cap B_i|$. We can isolate the $O(\log^\varepsilon n)$ term and apply the recursive inequality from Section 3.1. By linearity of expectation, the resulting running time after preprocessing is then $O(n\log^\varepsilon n)$ in expectation which concludes the theorem. ◀

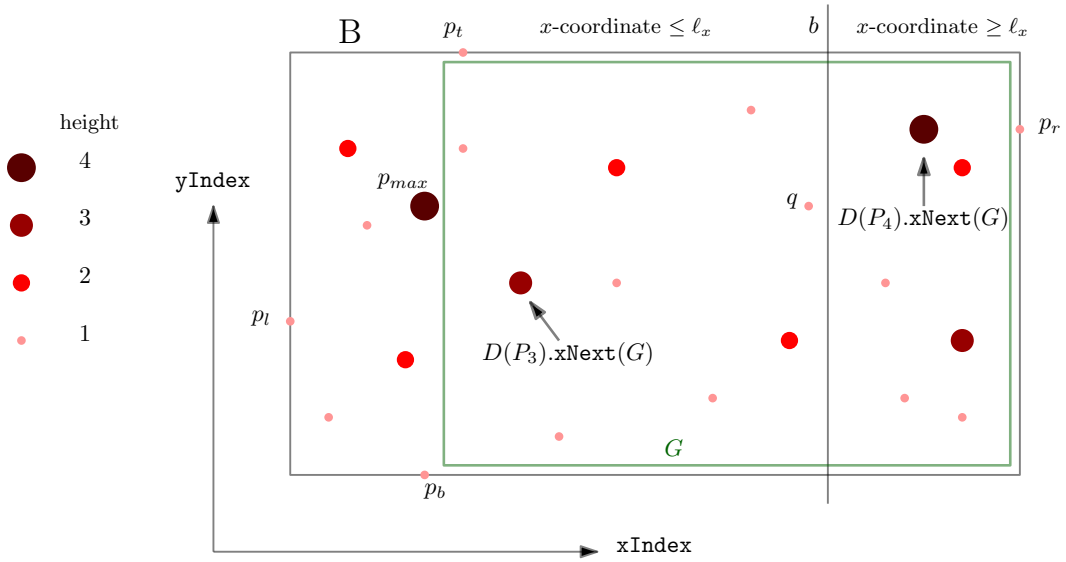

Figure 2 Visualization of our main algorithm. The auxiliary data structures $D(R_{P_i})$ for $i = 0, 1, 2, \ldots$ are used to implicitly represent a skip list.

▶ **Corollary 6** (Equivalence from [30]). *Let P be a size-n point set given as (A_x, A_y, π) where A_x and A_y sort P by x and y respectively, and π is the permutation that maps A_x to A_y. We can construct any proximity structure on P in expected $O(n\sqrt{\log n})$ time.*

5 Orthogonal Range Successor for the Real RAM

In Section 4 we show a Real RAM algorithm that computes a quadtree (and from it, any proximity structure) in expected $O(n\sqrt{\log n})$ time. A central ingredient of our algorithm is a data structure that solves the *orthogonal range successor problem* over some integer-valued data in the range $[n]$. To this end, we wish to make use of an existing Word RAM data structure by Belazzougui and Puglisi [7]. However, since we are on the Real RAM, we cannot immediately deploy such data structures. Instead we show that, by making use of a careful analysis of the bit complexity, the Word RAM operations used by this data structure can be simulated on Real RAM at no asymptotic overhead.

▶ **Theorem 3.** *Let π be a size-n integer array that specifies a permutation from $[n]$ to $[n]$. Let $I_\pi = \{(i, \pi(i)) : i \in [n]\}$ be the induced point set in $[1, n]^2$. On a Real RAM we can, with $O(n\sqrt{\log n})$ preprocessing, store I_π in a linear-size data structure that can answer the following queries in $O(\log^\varepsilon(n))$ time (for any fixed $\varepsilon > 0$):*

- *`xNext(rectangle` $R \subset [1, n]^2$`)` reports the leftmost and rightmost point in $I_\pi \cap R$.*
- *`yNext(rectangle` $R \subset [1, n]^2$`)` reports the topmost and bottommost point in $I_\pi \cap R$.*

Proof. Belazzougui and Puglisi [7, Theorem 4] show how to do exactly this in the Word RAM model using at most $\log n + O(1)$ bits per word. Therefore, it would suffice to simulate this Word RAM data structure on Real RAM with constant overhead per operation, and $O(n)$ preprocessing time. We will do exactly this. Note that we only need to support these operations on integers consisting of at most $\log n + O(1)$ bits.

We use the textbook Word RAM definition by Aho, Hopcroft, and Ullman [3]. Of these operations, the one we cannot immediately support is integer division. We will instead start by supporting a simpler operation: Rightward bit shifts by a fixed amount k (to be chosen later), or, equivalently, integer division by a fixed power of two 2^k. We denote this operation as $i >> k := \lfloor \frac{i}{2^k} \rfloor$. To support this operation, we will employ a variant of a standard trick used within Word RAM algorithms to support additional types of operations: An operation-specific lookup table. Let N be the maximum integer value that can be represented in $\log n + O(1)$ bits. Our lookup table will have size exactly N, the ith entry of which contains the result of $i >> k$. We will generate this lookup table in two phases:

- In Phase 1, we compute every multiple of 2^k, and mark the answers in the table.
- In Phase 2, we sweep through the table, and fill the remaining entries accordingly.

This pre-computation takes $O(N) = O(n)$ time to compute.

Now, we can use the fixed right-shift operation as follows: For any integer $i \leq N$ in a single register in Real RAM, we can now split it into $\lceil \frac{\log n + O(1)}{k} \rceil$ contiguous registers each containing a sequence of chunks with k bits each, in time proportional to this number of contiguous registers. To do this with an integer i, simply compute $i - (i >> k) * 2^k$ to get the lowest chunk of k bits, and then take $i \leftarrow i >> k$ to delete these extracted bits. These registers can be easily re-combined into one integer using multiplication and addition.

Now, to support integer division, we can employ a more standard variant of the same trick. First, we will brute-force a look-up table for small values, including remainder results. Second, we will essentially combine these small computations on chunked integers using basic long division. Pick $k = \frac{\log N}{3}$. We can brute-force compute a table that allows us to do integer

division on integers of size $\leq k$. We can afford to do each of these computations bit-by-bit (so $O(k^2)$ time per operation), since there is a small number of combinations ($O(2^{2k})$ in this case). We can do the same to compute modulo on integers of size $\leq k$. We can then combine the results using standard big-integer division methods (e.g., Knuth's Algorithm D [27]). ◀

Given integer division, all other typical Word RAM operations with a constant number of operands (such as bitwise `AND`, `OR`, `XOR`, and unary operations like `MSB`) can be simulated [23]. The construction is quite similar to the above proof. This is the standard approach used for extending the operation set in Word RAM.

6 Conclusion and Open Problems

In the appendix, we further populate our hierarchy. We recall a folklore result that triangulations are `1-Presortable`. We then show that we can adapt our techniques to construct a quadtree to construct a KD-tree instead, making KD-trees `2-Presortable`. We further expand the set of `2-Presortable` problems by showing that Orthogonal Segment Intersection Detection and computing the Maximum Empty Circle are `2-Presortable`. These latter two results use standard geometric techniques, but nicely illustrate the potential of this new class of geometric problems. Finally, we show in the full version that an onion-layer decomposition is `Presort-Hard` by constructing a class of labelled point sets with the same x- and y-order, that emits $\Omega((n/4)!)$ distinct combinatorial onion layers. Thus, there exists an $\Omega(n \log n)$-time information-theoretic lower bound even if the input is presorted. This paper thereby provides a comprehensive overview of a new hierarchy for geometric problems. As a by-product, we positively answer a long-standing open question regarding the fast construction of proximity data structures under presorted input – when randomisation is permitted.

We finalise our conclusion by posting several open problems. The most approachable open question appears to be derandomising our main result for constructing Quadtrees. We note that this is the only barrier to derandomising the proximity structures: NN-graphs, Delaunay Triangulations, Euclidean MSTs, Voronoi diagrams, and finding the Maximum Empty Circle, since the reductions for these problems are deterministic [30].

Another interesting open question is whether the time complexity for constructing Quadtrees (and hence all the other problems as well) can be improved to $o(n\sqrt{\log n})$. The main barrier of our construction is the use of orthogonal range predecessor search, which currently requires $O(n\sqrt{\log n})$ pre-computation time, even on Word RAM. An improved result for orthogonal range predecessor search on Word RAM with the same word-size properties would hence imply an improved result on Real RAM for all of these problems.

We also ask whether we could obtain a similar result for Quadtrees or other problems in higher dimensions. We note that one possible avenue for this would be to construct a data structure for the Range Minimum Query problem over points of magnitude at most $O(n)$ in d-dimensions (for $d > 2$), with preprocessing time $o(n \log n)$ and query time $o(\log n)$.

References

1 Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-Optimal Geometric Algorithms. *Journal of the ACM*, 64(1), March 2017. doi:10.1145/3046673.

2 Alok Aggarwal, Leonidas J. Guibas, James Saxe, and Peter W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete & Computational Geometry*, 4(6):591–604, 1989. doi:10.1007/BF02187749.

3 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974. ACM Digital Library ID: 10.5555/578775. URL: https://dl.acm.org/doi/10.5555/578775.

4 Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information processing letters*, 9(5):216–219, 1979. doi:10.1016/0020-0190(79)90072-3.

5 Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the Worst-Case Complexity of TimSort. *European Symposium on Algorithms (ESA)*, 112:4:1–4:13, 2018. doi:10.4230/LIPIcs.ESA.2018.4.

6 Xingjian Bai and Christian Coester. Sorting with predictions. *International Conference on Neural Information Processing Systems (NIPS)*, 2023. URL: https://dl.acm.org/doi/10.5555/3666122.3667277.

7 Djamal Belazzougui and Simon J Puglisi. Range predecessor and Lempel-Ziv parsing. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016. doi:10.1137/1.9781611974331.ch143.

8 Michael Ben-Or. Lower Bounds for Algebraic Computation Trees. *ACM Symposium on Theory of Computing (STOC)*, pages 80–86, 1983. doi:10.1145/800061.808735.

9 Kevin Buchin and Wolfgang Mulzer. Delaunay Triangulations in O(sort(n)) Time and More. *Journal of the ACM (JACM)*, 58(2), April 2011. doi:10.1145/1944345.1944347.

10 Sergio Cabello, Timothy M. Chan, and Panos Giannopoulos. Delaunay Triangulations with Predictions. *Innovations in Theoretical Computer Science Conference (ITCS)*, 362:31:1–31:23, 2026. doi:10.4230/LIPIcs.ITCS.2026.31.

11 Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël Jungers, and J. Ian Munro. Sorting under partial information (without the ellipsoid algorithm). *Combinatorica*, 33(6):655–697, December 2013. doi:10.1007/s00493-013-2821-5.

12 Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 161–173, 2010. doi:10.1137/1.9781611973075.15.

13 Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(3):485–524, 1991. doi:10.1007/BF02574703.

14 L. P. Chew and S. Fortune. Sorting helps for Voronoi diagrams. *Algorithmica*, 18(2):217–228, June 1997. doi:10.1007/BF02526034.

15 Sarita de Berg, Ivor van der Hoog, Eva Rotenberg, Daniel Rutschmann, and Sampson Wong. Instance-Optimal Imprecise Convex Hull. *European Symposium on Algorithms (ESA)*, 351:25:1–25:15, 2025. doi:10.4230/LIPIcs.ESA.2025.25.

16 Olivier Devillers. Delaunay Triangulation of Imprecise Points: Preprocess and Actually get a Fast Query Time. *Journal of Computational Geometry (JoCG)*, 2011. doi:10.20382/jocg.v2i1a3ff.ffinria-00595823.

17 Hristo Djidjev and Andrzej Lingas. On computing Voronoi Diagrams for Sorted Planar Point Sets. *International Journal of Computational Geometry & Applications*, 05(03):327–337, 1995. doi:10.1142/S0218195995000192.

18 David Eppstein, Michael T. Goodrich, Abraham M. Illickan, and Claire A. To. Entropy-Bounded Computational Geometry Made Easier and Sensitive to Sortedness. *Canadian Conference on Computational Geometry (CCCG 2025)*, August 2025. doi:10.48550/arXiv.2508.20489.

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

20 William Evans and Jeff Sember. The Possible Hull of Imprecise Points. *Canadian Conference on Computational Geometry (CCCG)*, 2011. URL: https://www.cs.ubc.ca/~will/papers/possiblehull.pdf.

21 Esther Ezra and Wolfgang Mulzer. Convex Hull of Points Lying on Lines in $o(n \log n)$ Time after Preprocessing. *Computational Geometry*, 2013. doi:10.1016/j.comgeo.2012.03.004.

22 Michael L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1(4):355–361, April 1976. doi:10.1016/0304-3975(76)90078-5.

23 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, December 1993. doi:10.1016/0022-0000(93)90040-4.

24 R.L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972. doi:10.1016/0020-0190(72)90045-2.

25 Bernhard Haeupler, Richard Hladík, John Iacono, Václav Rozhoň, Robert E. Tarjan, and Jakub Tětek. Fast and Simple Sorting Using Partial Information. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3953–3973, 2025. doi:10.1137/1.9781611978322.134.

26 Martin Held and Joseph Mitchell. Triangulating Input-constrained Planar Point Sets. *Information Processing Letters (IPL)*, 2008. doi:10.1016/j.ipl.2008.09.016.

27 Donald E Knuth. *The art of computer programming: Seminumerical algorithms, volume 2*. Addison-Wesley Professional, 2014. doi:10.1137/1012065.

28 Drago Krznaric and Christos Levcopoulos. Computing a threaded quadtree from the Delaunay triangulation in linear time. *Nordic Journal of Computing*, 5(1):1–18, March 1998.

29 D. Lee and Ying-Fung Wu. Geometric complexity of some location problems. *Algorithmica*, 1:193–211, November 1986. doi:10.1007/BF01840442.

30 Maarten Löffler and Wolfgang Mulzer. Triangulating the Square and Squaring the Triangle: Quadtrees and Delaunay Triangulations are Equivalent. *SIAM Journal on Computing*, 41(4):941–974, 2012. doi:10.1137/110825698.

31 Maarten Löffler and Wolfgang Mulzer. Unions of Onions: Preprocessing Imprecise Points for Fast Onion Decomposition. *Journal of Computational Geometry (JoCG)*, 2014. doi:10.1007/978-3-642-40104-6_42.

32 Maarten Löffler and Benjamin Raichel. Preprocessing Disks for Convex Hulls, Revisited. *arXiv preprint arXiv:2502.03633*, 2025. doi:10.48550/arXiv.2502.03633.

33 Maarten Löffler and Jack Snoeyink. Delaunay Triangulation of Imprecise Points in Linear Time after Preprocessing. *Computational Geometry*, 2010. doi:10.1016/j.comgeo.2008.12.007.

34 Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig. Checking geometric programs or verification of geometric structures. *Symposium on Computational Geometry (SoCG)*, pages 159–165, 1996. doi:10.1145/237218.237344.

35 Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012. doi:10.1007/978-1-4612-1098-6.

36 William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. doi:10.1145/78973.78977.

37 Raimund Seidel. A Method for Proving Lower Bounds for Certain Geometric Problems. *Computational Geometry*, 2:319–334, 1985. doi:10.1016/B978-0-444-87806-9.50017-7.

38 Michael Ian Shamos and Dan Hoey. Geometric intersection problems. *Symposium on Foundations of Computer Science (FOCS)*, pages 208–215, 1976. doi:10.1109/SFCS.1976.16.

39 Godfried T. Toussaint. Computing Largest Empty Circles with Location Constraints. *International Journal of Computer & Information Sciences*, 12(5):347–358, 1983. doi:10.1007/BF01008046.

40 Ivor van der Hoog, Irina Kostitsyna, Maarten Löffler, and Bettina Speckmann. Preprocessing Ambiguous Imprecise Points. *International Symposium on Computational Geometry (SoCG)*, 2019. doi:10.4230/LIPIcs.SoCG.2019.42.

41 Ivor van der Hoog, Irina Kostitsyna, Maarten Löffler, and Bettina Speckmann. Preprocessing Imprecise Points for the Pareto Front. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2022. doi:10.1137/1.9781611977073.122.

42 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. A Combinatorial Proof of Universal Optimality for Computing a Planar Convex Hull. *European Symposium on Algorithms (ESA)*, 351:102:1–102:13, 2025. doi:10.4230/LIPIcs.ESA.2025.102.

43 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Simpler Optimal Sorting from a Directed Acyclic Graph. *Symposium on Simplicity in Algorithms (SOSA)*, 2025. doi:10.1137/1.9781611978315.26.

44 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Tight Universal Bounds for Partially Presorted Pareto Front and Convex Hull. *arXiv:2512.06559 [cs.DS]*, 2025. preprint. doi:10.48550/arXiv.2512.06559.

45 Ivor van der Hoog and Daniel Rutschmann. Tight bounds for sorting under partial information. *Symposium on Foundations of Computer Science (FOCS)*, 2024. doi:10.1109/FOCS61266.2024.00131.

46 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. *Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975. doi:10.1109/SFCS.1975.26.

47 Marc van Kreveld, Maarten Löffler, and Joseph Mitchell. Preprocessing Imprecise Points and Splitting Triangulations. *SIAM Journal on Computing (SICOMP)*, 2010. doi:10.1137/090753620.

A Expanding the Population of our Hierarchy

In this section, we populate our hierarchy further by classifying several geometric problems. First, we show that computing a triangulation is `1-Presortable`. Next, we show that computing a KD-tree, the orthogonal segment intersection detection problem, and the maximum-area empty circle problem are `2-Presortable`. We do this by showing that they are not `1-Presortable` and providing randomised $o(n \log n)$-time algorithms that takes presortings as input. In the full version, we additionally show that several problems are `Presort-Hard`.

A.1 Triangulations are `1-Presortable`

We consider the problem of finding *some* triangulation of a planar point set. Although both the Triangulation and Delaunay Triangulation problems have tight $\Theta(n \log n)$ algorithms in general, it turns out that Triangulation is easier than Delaunay Triangulation in the context of our presorting hierarchy. For triangulations we assume the input to lie in general position to avoid degeneracies:

▶ **Theorem 7.** *Let P be a size-n point set in general position where no two points share an x-coordinate. Given an array A_x that stores P in sorted order, we can compute a triangulation of P in linear time.*

Proof. Since the input is sorted by x-coordinate, we can compute the edges of the convex hull of P in linear time [4]. Since no two points have the same x-coordinate, A_x also forms an x-monotone polygonal chain. If we take the union of the edges in each of these structures, we obtain a connected plane graph. For each face of this plane graph, we can then triangulate it with Chazelle's linear-time triangulation algorithm for simple polygons [13]. See Figure 3 for an example of this construction. ◀

A.2 Constructing a KD-tree

A KD-tree is a planar data structure that can be defined as follows: The root node represents a point set P. It splits P by x-coordinate along the point with median x-coordinate. Its child nodes split their remaining points by median y-coordinate instead and this recurses until the leaves store a single point. We show that a KD-tree is in the class of `2-Presortable` problems, employing a similar technique to that in Section 4:

▶ **Theorem 8.** *Let P be a size-n point set. Consider (A_x, A_y, π) where A_x and A_y sort P by x and y respectively, and π is the permutation that maps A_x to A_y. Given only A_x, there is a comparison-based $\Omega(n \log n)$ lower bound for constructing the KD-tree of P. Given (A_x, A_y, π), we can construct a KD-tree in expected $O(n\sqrt{\log n})$ time.*

Proof. First we show the lower bound: for a set of 1-dimensional points along the y-axis, a KD-tree is exactly a (balanced) binary search tree over the points. It follows that we can sort any set of points P by their y-value using a linear-time in-order traversal of the KD-tree of P. This gives an immediate reduction from sorting: given any set of values $V = (v_1, \ldots, v_n)$ create the point set $P = \{(0, v_i) \mid v_i \in V\}$. Then any order of V forms an array storing V sorted by x-coordinate. Yet, any $o(n \log n)$-time algorithm to construct a KD-tree implies an $o(n \log n)$-time algorithm for sorting V by the in-order traversal of the output.

Given a 2-presorting (A_x, A_y, π) we obtain an algorithm with a running time of expected $O(n\sqrt{\log n})$ by running the algorithm that is used in Theorem 5. Recall that we denote by $[1, n]^2 \cap \mathbb{N}^2$ the *rank space*. The point set P has a corresponding point set R_P in rank space where $(i, j) \in R_P$ if there exists a point $p \in P$ where $A_x[i] = p$ and $A_y[j] = p$.

In the proofs supporting Theorem 5, we first preprocess R_P by storing it in the orthogonal range successor data structure by Belazzougui and Puglisi [7] (abusing that coordinates in R_P are integers in $[n]$ to simulate this Word RAM algorithm on a Real RAM). Our algorithm then recursively halves a rectangle $R \subset \mathbb{R}^2$ by either a vertical line ℓ_x or horizontal line ℓ_y. Say we split on ℓ_x. We partition the points in $P \cap R$ along this line implicitly, using an implicit skip-list to find the rightmost point left of ℓ_x using exponential search. Our implicit skip-list algorithm executes the skip-list algorithm at $O(\log^{\varepsilon} n)$ overhead by querying the data structure to get the successor in the skip list. We then compare the successor to ℓ_x in constant time to test whether we need to go right or down in the skip list.

To construct a KD-tree, we can use a very similar procedure. Chan and Pătraşcu [12] show, on a Word RAM, how to preprocess a point set in $O(n\sqrt{\log n})$ time to answer orthogonal range counting queries in $O(\sqrt{\log n})$ time. In particular, their algorithm uses only $\log n$ bits (see the remark at the end of Section 1.3 in their paper). Hence, via our exact same technique, we can store R_P in this data structure also: simulating the Word RAM instructions on a Real RAM since R_P contains only coordinates in $[n]$. Whenever we want to split a rectangle R, we want to do so by the median x-coordinate (or y-coordinate). Via an orthogonal range counting query on $\Gamma(R)$, we know how many points there are in $P \cap R$. We now use almost the exact same exponential search as before, encountering points $p \in R$. However, instead of comparing p to a line we instead consider splitting $\Gamma(R)$ on the x-value of p. This corresponds to a rectangle $R' \subset R$ in the plane, and to a rectangle $G = \Gamma(R')$ in rank space. By performing an orthogonal range counting query on G, we count how many points there are in $R' \cap P$. This way, we can test whether p precedes, succeeds, or is the point with

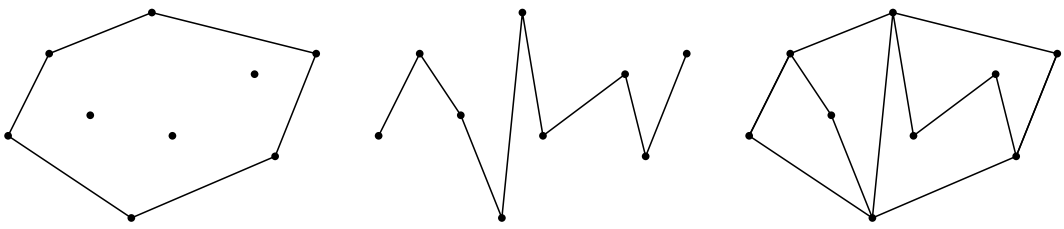

Figure 3 The construction for computing some triangulation, given A_x. We take the union of the convex hull (left) and an x-monotone polygonal chain through all the vertices (middle) to arrive at a connected plane graph whose vertices are exactly P (right).

median x-coordinate and thus we can execute our exponential search. Invoking this query takes $O(\sqrt{\log n})$ time per step in the skip-list (as opposed to the previous $O(\log^{\varepsilon} n)$ time). This does not asymptotically increase the overall running time. ◀

A.3 Orthogonal Segment Intersection Detection

For a set of horizontal and vertical line segments, we consider the problem of detecting an intersection between any pair of them. In this problem, we do not have a set of points, so it may not be immediately obvious what a presorting is. In this case, we consider a presorting of all the endpoints. This intuitively matches up with the standard $\Theta(n \log n)$-time algorithms for this problem and its generalizations [38], which all start by sorting the endpoints along the x-axis. We will take a related approach to prove the following result:

▶ **Theorem 9.** *Let S be a set of line segments and P be a size-n point set corresponding to all endpoints of the line segments. Consider (A_x, A_y, π) where A_x and A_y sort P by x and y respectively, and π is the permutation that maps A_x to A_y. Given only A_x, there is a comparison-based $\Omega(n \log n)$ lower bound for detecting if any two segments in S intersect. Given (A_x, A_y, π), we can detect if S contains any intersections in $O(n \log \log n)$ time.*

Proof. For the lower bound, we reduce from the element distinctness problem: We are given a multiset V of n real values and need to output whether V contains any duplicate elements. This problem is known to have an $\Omega(n \log n)$ comparison-based lower bound by Fredman [22].

We can construct a set S of line segments as follows: For the i'th element V with value v, create a horizontal line segment with endpoints $(0 + \varepsilon_i, v)$ and $(1 + \varepsilon_i, v)$, where $\varepsilon_i < 1$ is some arbitrary positive permutation. The result is a set of horizontal line segments, where the corresponding point set P even all has unique x-coordinates. Two values $v_1, v_2 \in V$ are identical if and only if their corresponding pair of horizontal line segments intersect.

For the upper bound, we consider as input a 2-presorting (A_x, A_y, π) of the endpoints P of segments in V. We assume that each value in A_x (and A_y) specifies a segment endpoint and which segment of S it belongs to. Observe that we can detect intersections between pairs of horizontal segments and pairs of vertical segments in $O(n)$ time: Simply bucket the horizontal segments by their y-coordinate in x-sorted order per-bucket, and then sweep through each bucket to detect intersections. Do the symmetric routine for the vertical segments.

What remains is to detect intersections between a pair of segments where one is horizontal and the other is vertical. We perform a line sweep from left-to-right, with events corresponding to each endpoint. At all times, we will maintain a data structure consisting of all the current horizontal segments, in order. We store the y-coordinates of these horizontal segments. In order to detect if a vertical segment intersects any of the current horizontal segments, it suffices to be able to perform predecessor and successor search on integers of size at most $O(n)$. This is exactly the task accomplished by a van Emde Boas tree [46] which uses $O(\log \log n)$ time per query and update. ◀

A.4 Maximum Empty Circle

The final problem that we consider is the maximum empty circle problem. Given a point set P, we wish to compute a maximum-area circle which has its centre in the convex hull of P but contains no point of P [39]. It is a classic geometric problem with an $O(n \log n)$ time upper bound and a matching comparison-based lower bound.

▶ **Theorem 10.** *Let P be a size-n point set. Consider (A_x, A_y, π) where A_x and A_y sort P by x and y respectively, and π is the permutation that maps A_x to A_y. Given only A_x, there is a comparison-based $\Omega(n \log n)$ lower bound for computing a maximum empty circle of P. Given (A_x, A_y, π), we can compute a maximum empty circle in expected $O(n\sqrt{\log n})$ time.*

Proof. For the lower bound, we reduce from the *largest gap* problem. In this problem, the input is a set of n values V. Consider the set $S = (s_1, \ldots, s_n)$ which sorts V. The maximum gap is the index i that maximizes $s_{i+1} - s_i$. This problem was shown to have a comparison-based $\Omega(n \log n)$ lower bound by Lee and Wu [29, Lemma A.2]. Moreover, as the proof is combinatorial it still holds even if we assume some minimal separation between the values in V. So let V be a set of values where all values differ by at least 1. Denote by S its unknown sorted set. Given V, we construct in linear time a point set P by constructing for each point $v_i \in V$ two points: $(-\varepsilon_v, v)$ and $(+\varepsilon_v, v)$ where $\varepsilon_v < 0.001$ is some arbitrary value. The result is a column of 4-gons whose width is less than 0.001 and whose height corresponds to the gaps in S. Since the x-coordinates are arbitrary, an array A_x that sorts P by x-coordinates gives zero additional information. The convex hull of these points is a rectangle of width 0.001 and the largest disk that has its centre in this rectangle (but contains no point of P) has a diameter which corresponds to the largest gap in S. Given a maximum empty circle, we can find its closest four points in P in linear time and thus the values in V that realise the maximum gap in S.

For the upper bound, Toussaint [39] showed an algorithm that given the Voronoi diagram of a point set computes the maximum empty circle in linear time. By Corollary 6, given (A_x, A_y, π), we can compute the Voronoi diagram of P in expected $O(n\sqrt{\log n})$ time, which concludes the proof. ◀

B Recursive Inequality

The central algorithm in our paper is analyzed using the following recursive inequality. Consider a recursive procedure, that operates over an array of size N. For convenience of notation, let in this section denote $[a, b] := \{a, a+1, \ldots, b\}$. When called on a subarray $[a, b]$ of size $n := b - a + 1$, the procedure selects some splitting index $a \leq i < b$, and is afterwards called on the subarrays $[a, i]$ and $[i + 1, b]$. The splitting index i depends on a, b in some arbitrary fashion. We are interested in the runtime $T(N)$ of the procedure. Let $n := b - a + 1$ and $n_1 := i - a + 1$ and $n_2 = b - i$ be the sizes of the involved subarrays. We assume that during each recursive call the additional work done by the procedure depends only on the *minimum* of n_1, n_2 times a small multiplicative overhead factor M. That is, we have $T(1) = O(1)$ and for some function $f : \mathbb{N} \to \mathbb{N}$ and some $M > 0$ for all $n > 1$

$$T(n) \leq T(n_1) + T(n_2) + M \cdot f(\min(n_1, n_2)). \tag{2}$$

In our main theorem, we will let $M = \log^{\varepsilon} N$. A technical drawback of the above equation is that we can only consider such functions where each recursive call handles one contiguous subarray. For our main theorem, we require a slightly more general setting. Namely, assume that $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is a function that satisfies $T(1) = O(1)$ and for all $n > 1$

$$T(n) \leq \max_{1 \leq i \leq n-1} T(i) + T(n - i) + Mf(\min(i, n - i)). \tag{3}$$

The following lemma is a known fact [9], but we repeat its proof here for convenience of the reader. The main insight is that assuming f is strictly sublinear, the runtime $T(n)$ is surprisingly low, no matter how unbalanced the splitting indices are chosen.

▶ **Lemma 11.** *If $T(n)$ satisfies (2) or (3) and $f(x) \leq Cx^{\alpha}$ for some constants $0 < C, 0 < \alpha < 1$, then $T(n) \leq M \cdot O(n)$.*

Proof. First, we argue that we can w.l.o.g. only consider some function $T(n)$ defined over contiguous subarrays such that (2) is satisfied. We claim that this implies the more general case of (3) as well. Indeed, assume we have some function $T(n) : \mathbb{N} \to \mathbb{R}_{\geq 0}$ that satisfies (3). For each n, there exists some i_n that maximizes the right-hand-side of (3). We imagine the recursive procedure T' operating over the array $[1, N]$, such that whenever this procedure is called on some subarray $[a, b]$ of size $n = b - a + 1$, it decides which splitting index to select only based on n. Namely, it selects splitting index $a + i_n - 1$. One can then prove by induction that T' is an upper bound for T and satisfies (2). Hence it suffices to prove our theorem only for recursive procedures defined on contiguous subarrays satisfying (2).

Let now $T(n)$ be some function that satisfies (2). We imagine the procedure running on the complete array $[1, N]$. Every time the procedure is called on some subarray $[a, b]$, we have to perform work $Mf(\min(n_1, n_2))$ where n_1, n_2 are determined by the splitting index i corresponding to that subarray $[a, b]$. We consider a charging argument. If $n_1 < n_2$, we charge the total work $Mf(n_1)$ to the left half $[a, i]$ of the array. More precisely, we charge each of the n_1 individual elements of $[a, i]$ with a charge of $Mf(n_1)/n_1$. In the other case $n_2 \leq n_1$, we distribute the total work $Mf(n_2)$ by charging each of the n_2 individual elements of $[i + 1, b]$ with a charge of $Mf(n_2)/n_2$. Let $\mathcal{S}$ be the set of all subarrays that are charged when computing the function for the complete array $[1, N]$, and let for each subarray $S \in \mathcal{S}$ denote ℓ_S the number of elements of that subarray. Then, by the above explanation

$$T(N) = \sum_{S \in \mathcal{S}} Mf(\ell_S) = \sum_{S \in \mathcal{S}} \sum_{i=1}^{\ell_S} Mf(\ell_S)/\ell_S.$$

We now count this quantity in a different way. Consider some element $j \in [1, N]$ that was charged at least once and let $\mathcal{S}(j) := \{S \in \mathcal{S} : j \in S\}$ be the set of corresponding subintervals that were charged whenever j was charged. W.l.o.g. let $\mathcal{S}(j) = \{S_0, \ldots, S_k\}$ for some k, where $\ell_{S_k} \geq \ell_{S_{k-1}} \geq \cdots \geq \ell_{S_0}$. Note that every time j gets charged, it must be on the smaller side of the corresponding subinterval. This means that if j is charged twice in succession, its subinterval must shrink by a factor of 2 or more. In other words, $\ell_{S_i} \geq 2\ell_{S_{i-1}}$ for all $i = 1, \ldots, k$. Since $\ell_{S_0} \geq 1$ we have $\ell_{S_i} \geq 2^i$. Then we can compute the total charge by summing the contribution of each element j.

$$\begin{aligned} T(N) = \sum_{j=1}^{N} \sum_{S \in \mathcal{S}(j)} Mf(\ell_S)/\ell_S \qquad &\leq \sum_{j=1}^{N} \sum_{S \in \mathcal{S}(j)} MC\ell_S^{\alpha-1} \leq \sum_{j=1}^{N} \sum_{i=0}^{\infty} MC2^{i(\alpha-1)} \\ &= \sum_{j=1}^{N} MC\frac{1}{1 - 2^{\alpha-1}} = M \cdot O(N). \end{aligned}$$

This was to show. ◀

▶ **Corollary 12.** *If $T(n)$ satisfies (2) or (3), and f is the runtime of a randomized procedure with $\mathbb{E}[f(x)] \leq Cx^\alpha$, then $\mathbb{E}[T(n)] \leq M \cdot O(n)$.*

Proof. This follows from Lemma 11 and the linearity of expectation. More precisely, even if the running times $T(n)$ and $f(x)$ are random variables, it is still true that $T(n) = \sum_{S \in \mathcal{S}} Mf(\ell_S)$. Then we have

$$\mathbb{E}[T(N)] = \sum_{S \in \mathcal{S}} M\mathbb{E}[f(\ell_S)] \leq \sum_{S \in \mathcal{S}} MC\ell_S^\alpha \leq M \cdot O(N),$$

where the last inequality follows from the same charging argument, and is true, no matter which splitting indices are chosen. ◀