Fast Computation of Output-Sensitive Maxima in a Word RAM

Peyman Afshani [*]
Department of Computer Science
Aarhus University
peyman@madalgo.au.dk

November 6, 2013

Abstract

In this paper, we study the problem of computing the maxima of a set of n points in three dimensions with integer coordinates and show that in a word RAM, the maxima can be found in $O\left(n \log\log_{n/h} n\right)$ deterministic time in which h is the output size. For $h = n^{1-\alpha}$ this is $O(n \log(1/\alpha))$. This improves the previous $O(n \log\log h)$ time algorithm and can be considered surprising since it gives a linear time algorithm when $\alpha > 0$ is a constant, which is faster than the current best deterministic and randomized integer sorting algorithms. We observe that improving this running time is most likely difficult since it requires breaking a number of important barriers, even if randomization is allowed.

Additionally, we show that the same deterministic running time could be achieved for performing n point location queries in an arrangement of size h. Finally, our maxima result can be extended to higher dimensions by paying a $\log_{n/h} n$ factor penalty per dimension. This has further interesting consequences for example it preserves the linear running time when $h \le n^{1-\alpha}$, for a constant $\alpha > 0$, and thus it shows that for a variety of input distributions the maxima can be computed in linear expected time *without* knowing the distribution.

1 Introduction

For two points p and q in d dimensions, we say p *dominates* q if and only if each coordinate of p is greater than that of q. The *maxima* of a set P of n points is the subset of points that are not dominated by any point in P.

Computing the maxima is one of the classical and fundamental problems in computational geometry and not surprisingly, it has been studied extensively. The problem arises naturally in many applications such as multi-criteria optimization and it has a wide range of applications. For instance, when comparing cars on gas efficiency, price, and rating, if each of the three categories for a car A is better than that of a car B, then the two cars can be represented by points in three-dimensional space where car A dominates car B which means points on the maxima represent the "best" cars. This example shows that the dominance relation is a natural generalization of ordinary comparison to higher dimensions.

Previous Results. The first algorithm to compute the maxima in two and three dimensions was given by Kung et al. [18], in 1975, and it runs in $O(n \log n)$ time (see also [5]). The result extends to higher dimensions by paying a $\log n$ factor penalty for each added dimension. They also proved a lower bound of $\Omega(n \log n)$ in the binary comparison tree model matching the complexity of the algorithm in two and three dimensions. In two and three dimensions, output-sensitive results were obtained by Kirkpatrick and Seidel in 1985 [17]. Their algorithm runs in $O(n \log h)$ time where h is the output size. Once again, in the algebraic decision tree model,

[*]Center for Massive Data Algorithmics (MADALGO), a Center of the Danish National Research Foundation.

this running time is optimal (also see [16] for another lower bound proof). Finally, if the input has integer coordinates, then it is known that the maxima can be computed deterministically in $O(n \log \log n)$ time. Using standard techniques [8], this gives an output-sensitive algorithm with running time of $O(n \log \log h)$. Note that since the best deterministic sorting algorithm runs in $O(n \log \log n)$ time, improving the general $O(n \log \log n)$ algorithm at least requires breakthrough results in sorting.

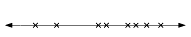
In addition to the classical results mentioned above, the maxima problem has been a favorite when studying new paradigms of computation beyond the worst-case analysis. The most popular approach is to consider the case when the input points come independently from a distribution, (there are many results in this category, e.g., see [6, 14]). However, the maxima has also been studied in many other models such as, the distribution-sensitive model [21], the instance-optimal model [1], self-improving model [13], and under various uncertainty assumptions (see e.g., [20]).

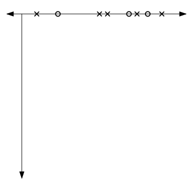
Our Results. In light of the overwhelming attention given to this problem, one might think that anything that could be said for the classical output-sensitive 3D maxima problem has already been said. Thus, it was a great surprise for us to realize that the known output-sensitive algorithm for the maxima problem is not optimal in the word RAM model, specially, since this is the most natural model for the problem given the orthogonal nature of the problem. In this paper, we show that the maxima of a set of three-dimensional points with integer coordinates in which each coordinate is w bits long can be computed in $O(n \log \log_{n/h} n)$ time deterministically in a word RAM with word size w. For $h = n^{1-\alpha}$ this is $O(n \log(1/\alpha))$ which is linear if α is a constant. We also observe that improving this bound, even by allowing randomization, requires breakthrough results in a number of important problems. The algorithm can be generalized to higher dimension by paying a $\log_{n/h} n$ factor penalty for each added dimension. Along the way, we also obtain other interesting results. We show that n offline point location queries on an arrangement of size h can be done in $O(n \log \log_{n/h} n)$ time.

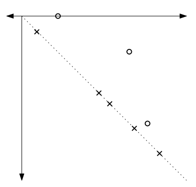
Relevance to Results on Random Pointsets. There are many results that deal with computing the maxima (and the related problems such as the convex hull) under the assumption that the input points are drawn independently from a probability distribution. In many cases in fact it is possible to compute the maxima in linear time [14, 12, 4, 7]. However, such results use some special properties of the distribution and thus the resulting algorithms are very much tailored to only work with that distribution. For example, an algorithm presented by Bentley, Clarkson, and Levine [4] concerns the case when each coordinate of the input points is drawn uniformly from an interval (say the unit interval). In this case, they show that with high probability one point will dominate almost all the other points so most points can be easily pruned. That one point itself can be found in a straightforward way by looking at each coordinate and locating the $O(\log n)$ points with the largest coordinate. Our algorithm on the other hand, does not require the knowledge that the input is drawn from a distribution. As long as the output is not too large (which is the case for all the distributions considered), the maxima can be found in linear time.

Our Techniques. The main tool that we borrow from the previous literature is Han and Thorup's linear time $\sqrt{n}$-way splitting algorithm [15]. However, non-trivial amount of work is required to apply this tool to a three-dimensional problem, specially since we have very little time to process each point and we cannot use randomization to prune points. To circumvent these barriers, we use a non-traditional sweep-plane approach where sweep order on the z-axis is determined by applying the splitting algorithm. The sweep is based on a two-dimensional dynamic problem with very fast query time but very low update time. The high-level idea is that since the output size is h, we can afford $O(n/h)$ update time for the dynamic problem that will help us reach very fast query times.

In the next section, we explore the basics both to warm-up for the more technical parts of the paper and also to get introduced to the important barriers that are in the way of getting a better result. In Section 3 we present our algorithm to compute the maxima. Finally, in Section 4, we consider the offline point location problem, and the extension of our algorithm to higher dimensions.


(a) Input of n integers.


(b) Random sample marked by circles.


(c) The 2D instance.

2 Preliminaries and Barriers

The input to the maxima problem is a set P of n points. The output is the subset $M \subset P$ of all the points that are not dominated by any point in P. We denote the size of M by h. Notice that the points in M can be returned in any order. Furthermore, for every point $p \in P \setminus M$, we require the algorithm to output a *certificate*, that is, a point q, $(q \neq p)$ that dominates p; q is not required to be a maxima point. Thus, the output size is $O(n)$.

As discussed, in the comparison and the Algebraic decision tree models, the status of the problem in two and three dimensions is fully settled: $O(n \log n)$ in the worst-case and $O(n \log h)$ for outputs of size h. By using a technique due to Chan [8], it is possible to use an algorithm that runs in $nf(n)$ time, and build an output-sensitive algorithm with running time of $O(nf(h))$. In RAM, computing the maxima in $O(n \log \log n)$ time is folklore and combined with Chan's method, this gives an $O(n \log \log h)$ time algorithm. Since the best known deterministic sorting algorithm runs in $O(n \log \log n)$ time, there is little hope to compute the maxima in $o(n \log \log n)$ time without finding a faster sorting algorithm. It might thus sound reasonable to assume that the $O(n \log \log h)$ should have the same barrier. As it turns out this is not true.

The crucial point is that the complexity of the planar output-sensitive maxima is not tied to sorting but to *h-way splitting*, which is the following problem: given a set X of n integers and a parameter h, partition X into h sets $X_1, \dots, X_h$ of roughly equal size such that all the elements of X_i are smaller than those in X_{i+1}. The fact that in the word RAM model there is a surprisingly fast splitting algorithm gives us motivation to tackle the output-sensitive maxima problem. The following theorem is the main tool that we borrow from the literature.

Theorem 1. *[15] In a word RAM, $\sqrt{n}$-way splitting can be done deterministically in linear time. Also, h-way splitting can be done deterministically in $O(n \log \log_{n/h} n)$ time.*

As discussed, in two dimensions, the connection between splitting and output-sensitive maxima problem is quite strong. The following observation follows from folklore techniques (e.g., see [17]).

Observation 1. *If h-way splitting can be done in $nf(n,h)$ time, then the planar maxima can be computed deterministically in $O(nf(n,h))$ time.*

Proof sketch. Do a $2h$-way splitting according to the value of the x-coordinate and let $P_1, \cdots, P_{2h}$ be the resulting subsets. For each subset P_i, find the point p_i with the largest y-coordinate. Observe that if a point p_i dominates a point p_j (obviously we must have $i > j$), then p_i dominates all the points in P_j. Since there are at most h maxima points and $2h$ sets, at least half of the sets can be pruned. Repeat until few points are left and then compute the maxima directly. □

Next, we observe that the maxima problem is at least as hard as the h-way splitting problem.

Lemma 1. *Assume there exists an algorithm that given a planar point set of size n, outputs the h maxima points in $nf(n,h)$ time together with a certificate for any pruned point. Then, h-way splitting can be done in $O(nf(n,h) + sort(h))$ expected time, in which $sort(h)$ is the time it takes to sort h integers.*

Proof. Let X be a set of n integers (Figure 1a). Randomly sample a subset I of h indices from $[n]$ (Figure 1b). Sort the elements corresponding to the indices, i.e., $x_{i_1} < \cdots < x_{i_h}$ in which $i_j \in I$. For any index $i \notin I$, define the point $(x_i, -x_i)$. For index i_j, define the point $(x_{i_j}, -x_{i_{j-1}})$ $(X_{i_0} = 0)$ (Figure 1c). It is easy to see that any non-maxima point has only one certificate and using the certificate, we can easily obtain an h-way splitting. $\square$ $\square$

The best randomized sorting algorithm runs in $O(n\sqrt{\log\log n})$ time which means if $f(n)$ is super-constant, a better splitting algorithm leads to a better sorting algorithm by the following observation.

Observation 2. *If h-way splitting can be done in $O(n(\log\log_{n/h} n)f^{-1}(n))$ time then integer sorting can be done in $O(n\sqrt{\log\log n f^{-1}(n)})$ expected time. Here, we assume $f(n) = \omega(1)$.*

Proof. The proof is exactly a rewrite of the technique used by Han and Thorup [15] (Section 2) using the additional parameter $f(n)$. $\square$ $\square$

We can summarize the discussions in this section as follows: First, unless one can prune the points without certificates, output-sensitive computation of maxima is at least as hard as h-way splitting. Second, coming up with an algorithm to compute the maxima that is always faster $O(n\log\log_{n/h} n)$ requires improving the best known randomized sorting algorithm. Third, even though sorting can be done in $o(n\log\log n)$ time using randomization, it is not know how to compute the maxima for $h = \Omega(n)$, in $o(n\log\log n)$ time in the worst-case. In other words, beating the worst-case bound of $O(n\log\log n)$ is still open even if randomization is allowed and even though randomized sorting can be accomplished in $O(n\sqrt{\log\log n})$ time.

With this discussion, it is clear that the $O(n\log\log_{n/h} n)$ bound is the best we can hope for, given the current status of the integer sorting problem. Since in two-dimensions obtaining such an algorithm is not too difficult, the main challenge is to obtain the aforementioned bound in three dimensions. We will do this in the next chapter.

3 3D Maxima

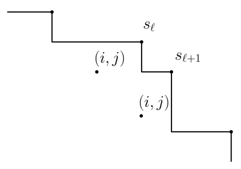
In this section we present our algorithm to compute the maxima of a set P of n points in three dimensions. We will assume that we know the output size, h. This assumption can be removed by "guessing" the output size similar to the previous output-sensitive algorithms [8]. Furthermore, we will assume that $h \le n/\log^4 n$. If $h \ge n/\log^4 n$ then our target running time becomes $O(n\log\log n)$ and we can use one of the previously known algorithms.

Our overall strategy is simple: we sweep a plane (from $z = +\infty$ to $z = -\infty$) to prune all but $O(n/\log n)$ of the points and then we run a classical algorithm on the rest of the points.

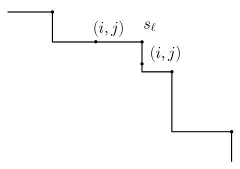
Our sweep-plane approach is rather non-traditional: normally, to perform a sweep approach, we need to sort the points by the z-coordinate but since we are essentially aiming for $o(\log\log n)$ processing time per point, we cannot afford that. Incidentally, this non-traditional sweep is also used by Kirkpatrick and Seidel in their output-sensitive maxima algorithm [17] but there approach does not directly work for us as there is no obvious way to perform one of their main pruning steps within our desired bound (specifically, step 2.1 in their "max3(T)" algorithm). To overcome this, we need additional ideas.

Using the h-way splitting algorithm, we reduce the universe size such that many points receive the same x, y, and z-coordinates. As a negative consequence, the reduction can alter the shape of the maxima region but it has the benefit that it helps us obtain a data structure for dynamic predecessor problem that has very fast queries (i.e., $o(\log\log n)$ query time) but with high update time ($O(n/h)$ update time). The hope is that even in the reduced universe, there would be $O(h)$ maxima points so that during the sweep we can get away with only $O(h)$ updates. We use more pruning techniques to make up for the damage done by the reduction in the universe size. The details of the algorithm are presented below.

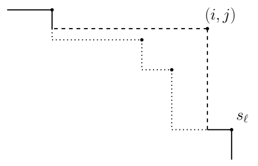
Reduction to Small Universe. Let $t = h\log n$. We begin the algorithm by a rank-reduction type technique. For each coordinate, we perform a t-way split, resulting in subsets $X_1, \dots, X_t$, $Y_1, \dots, Y_t$, and $Z_1, \dots, Z_t$, arranged in increasing order, that contain the x, y, and z-coordinates of the points respectively. Consider an input point p with coordinate (x, y, z) and assume $x \in X_i$, $y \in Y_j$, and $z \in Z_k$. We map p to the point $p^{(r)} = (i, j, k)$. The mapped points thus live in $[t]^3$, a much smaller universe. Let $P^{(r)}$ be the set of



(a) Cases when (i,j) is pruned.



(b) Cases when (i,j) is ignored.



(c) (i,j) triggers a Z-pruning and it will be inserted if it survives.

mapped points. We call $P^{(r)}$ the *reduced pointset*. Despite its inaccuracy, this transformation will preserve the dominance relation between many pairs of points and thus it will still allow us prune most of the points.

Sweep Order with a Twist. To prune the majority of the points, we follow a sweep-plane approach. We begin the sweep by processing all the points with z-coordinate t, and then the points with z-coordinate $t-1$, and we continue the sweep downward, until all the points are swept. Note that the sweep order comes directly from the previous step and the points with the same z-coordinate (i.e., the points in Z_i) can be swept in arbitrary order. This results in the following unusual situation: even if the z-coordinate of a point p is smaller than the z-coordinate of a point p', it is possible to sweep $p^{(r)}$ first and then later $p'^{(r)}$, however, this can only happen if both p and p' belong to the same Z_i bucket (i.e., mapped to the same z value). These out-of-order sweeps can cause a problem later on. To deal with them, we need special pruning steps that are discussed next.

Z-Pruning. To deal with the out-of-order sweeps, once in a while, we will issue a special pruning procedure that we call *Z-pruning*: given a parameter k, we simply go through all the points in Z_k and prune those dominated by a point in Z_k. In other words, if a point $p \in Z_k$ is dominated by another point $q \in Z_k$, we will delete p. We can do this in $O(|Z_k| \log |Z_k|) = O((n/t) \log n)$ time. Remember that Z-pruning is done using the original coordinates of the points and not the mapped coordinates. We need Z-pruning because without it, the number of updates into the data structure might significantly exceed h, an unpleasant outcome that can ruin our running time.

Dynamic Staircase. The sweep-plane approach corresponds to maintaining the maxima of a pointset (to be exact, the projection of the reduced pointset onto the xy-plane) in two dimensions under deletions and insertions. The two-dimensional maxima is in the form of a staircase. Let $s_1, \dots, s_r$ be the current staircase, sorted from left to right. Assume we are sweeping the point $p^{(r)} = (i, j, k)$, meaning, we need to insert (i, j) into this staircase. To do so, we find the successor, s_ℓ, of i, that is, we consider all the staircase vertices whose x-coordinate is greater than or equal to i and pick the one with the smallest x-coordinate to be s_ℓ. If (i, j) is dominated by s_ℓ (remember this requires that the x and y-coordinates of s_ℓ be strictly larger than i and j, respectively) then (i, j) is pruned. If this is not the case we check with $s_{\ell+1}$: if (i, j) is dominated by $s_{\ell+1}$ then (i, j) is pruned (see Figure 2a). If (i, j) lies on the boundary of the staircase (see Figure 2b) then we *ignore* (i, j); it will neither be inserted into the staircase nor will it be pruned and we will take care of these ignored points later. Otherwise, j is larger than the y-coordinate of s_ℓ, and thus (i, j) needs to be inserted as a new staircase vertex. However, before doing so, we issue a Z-pruning with parameter k, unless of course a Z-pruning with parameter k was already issued. We insert (i, j) into the staircase only if it survives the Z-pruning step. After the insertion, we check the existing staircase vertices that might be dominated by (i, j); these are deleted from the staircase (see Figure 2c).

Now we describe how to maintain the staircase under insertions and deletions. In general, this requires $\Omega(\log n / \log\log n)$ query time (reduction from dynamic marked ancestor problem [3]) but fortunately, ours is a special case and it permits a faster solution.

Staircase Data Structure. Our data structure to maintain the staircase is essentially a modification of van Emde Boas tree. Let $f = \sqrt{n/t}$. Build a tree T of fanout f on the universe $[t]$. The tree will have height

$\log_f t$. We allow the following operations: we can mark or unmark a leaf or given a query leaf, we can ask for the first marked leaf to the right of the query. Consider an internal node v with f children $v_1, \dots, v_f$. If the subtree at v contains at least one marked leaf, then for each v_i, we store the index of the smallest marked leaf to the right of the subtree of v_i and we call this *the successor of* v_i. This is stored for all the children of v, even those that do not contain any marked leaves.

Updates and Queries. It is not too difficult to see that updates can be handled in $O(f \log_f t)$ time, by walking down and up the tree and updating the relevant information. The successor queries now can be answered in $O(\log(\log_f t))$ time: Consider a query leaf. We access the ancestor v of the query leaf at depth $(\log_f t)/2$. Let v_i be the child of v that contains the query leaf. Both v and v_i can be found in constant time by address calculation. If the subtree at v does not contain any marked leafs, then we need to recurse on the top half of the tree. Otherwise, we have stored the successor of v as well as every child of v, including v_i. If the subtree v_i does not contain any marked leaf, then the answer to the query is the successor of v_i. Otherwise, after accessing the successor of v_i, we need to recurse on the subtree rooted at v_i. Either way, after spending $O(1)$ time, we will end up recursing on a tree of half the height, resulting in $O(\log\log_f t)$ query time.

Analysis. We first bound the number of times Z-pruning is invoked. This is important as Z-pruning is an expensive operation.

Lemma 2. *At most h calls are made to the Z-pruning procedure.*

Proof. Remember that we run our sweep on the reduced point set $P^{(r)}$. Also note that in our sweep approach, some points will be ignored while sometimes we will issue special pruning steps. Consider a point $p^{(r)} = (i, j, k)$ that when swept leads to the invocation of Z-pruning with parameter k. This can only happen if inserting (i, j) into the staircase would have resulted in the creation of a new staircase vertex. To prove the lemma, it suffices to show that Z_k contains at least one maxima point of P. If p is a maxima point of P then our claim is trivially true so assume p is dominated by a maxima point $m \in P$. By definition we have $p^{(r)} = (i, j, k)$ and let $m^{(r)} = (i', j', k')$. As m dominates p, we have $i \le i'$, $j \le j'$, and $k \le k'$. If $m \in Z_k$, then our claim is trivially true again. Thus, we must have $k < k'$ but this guarantees that m is swept before p, which implies when sweeping p, the point (i', j') must have been inserted into the staircase. Finally, observe that p will be ignored if $i = i'$ or if $j = j'$, meaning, Z_k contains a maxima point. □ □

Next lemma is quite natural sounding but it is not completely trivial due to the existence of ignored points.

Lemma 3. *Consider a staircase vertex $s = (i, j)$ and let A be the set of points p such that $p^{(r)} = (i', j', k')$ where $i' = i$ and $j' = j$. A contains at least one maxima point in the original point set.*

Proof. Let p be the point that causes the creation of the staircase vertex s (note that p may not be a maxima point). We must have $p^{(r)} = (i, j, k)$ for some $k \in [t]$. This means that a Z-pruning with parameter k was issued sometime in the past and p has survived the pruning. With a slight abuse of notation, redefine $A \subset P$ to be the subset of the points that are mapped to (i, j, k) that also have survived the Z-pruning step. We will show that no point $m \in P \setminus A$ can dominate a point in A. Assume to the contrary that there is a maxima point $m \in P \setminus A$ that dominates a point $a \in A$. By definition we have $a^{(r)} = (i, j, k)$ and let $m^{(r)} = (i', j', k')$. We know $i \le i'$, $j \le j'$, and $k \le k'$. We know a has survived the Z-pruning so we must have $k < k'$ but this implies m is swept before a. Since m is a maxima point, it cannot be pruned, meaning, either m must have been ignored or inserted into the staircase. Both of these options imply the existence of a staircase vertex $m' = (i'', j'')$ with $i' \le i''$ and $j' \le j''$ thus this results in a contradiction since the existence of m' will either lead to $p^{(r)}$ being pruned or ignored. □ □

Corollary 1. *At most h staircase vertices are created (or deleted) during the sweep.*

Finally, we show that only a small fraction of the points can be ignored.

Lemma 4. *In total, $O(n/\log n)$ points are ignored throughout the sweep approach.*

Proof. For a point p with $p^{(r)} = (i, j, k)$ to be ignored, there should be a staircase vertex $v = (i', j')$ with either $i = i'$ or $j = j'$. By Lemma 3, there is a maxima point p that gets mapped to (i', j'). It is clear that we either have $p \in X_{i'}$ or $p \in Y_{j'}$ and since each such set contains $O(n/t)$ points and we have h maxima points, the total number of ignored points is $O(hn/t) = O(n/\log n)$. $\square$ $\square$

Theorem 2. *The maxima of set of n points with w bit integer coordinates can be found in $O(n \log\log_{n/h} n)$ time deterministically, in a RAM with word size of w.*

Proof. First, we run the sweep algorithm outlined above which results in a (significant) subset of the points being pruned. Let P' be the subset of the points left (note that this includes the ignored points). We run a classical maxima algorithm on P'.

The correctness of the algorithm is obvious. To bound the running time, we need to show two things: first, that the sweep algorithm runs within the desired bound and second that the size of P' is not too large (we will prove that $|P'| = O(n/\log n)$).

We begin by analyzing the sweep algorithm. The reduction of the universe to $[t]^3$ can be done in $O(n \log\log_{n/t} n)$ time which is $O(n \log\log_{n/h} n)$. Each point that is swept issues one query to the modified van Emde Boas tree. However, by Corollary 1, at most h staircase vertices are created and deleted. This means, the total number of mark and unmark operations done on the tree is at most $2h$. The tree performs each query in $O(\log\log_f t) = O(\log\log_{\sqrt{n/t}}(h \log n)) = O(\log\log_{n/h} n)$ time. The total time spent performing the updates is $O(hf \log_f t) = O(h\sqrt{n/h} \log n) = O(n)$. By Lemma 2, there are at most h calls made to the Z-pruning procedure. Since the size of each set Z_k is $O(n/t)$, each such call is executed in $O(n(\log n)/t) = O(n \log n/(h \log n)) = O(n/h)$ time resulting in $O(n)$ total time.

Thus, it remains to bound the size of P'. By Lemma 4, there are $O(n/\log n)$ ignored points. Note that any other point is either pruned or is inserted into the staircase. Since there are at most h staircase vertices by Corollary 1, we have $|P'| = O(h + n/\log n) = O(n/\log n)$, which means, the maxima of P' can be found in $O(n)$ time. $\square$ $\square$

4 Higher Dimensions and Related Problems

In this section, we further explore the implications of our results.

4.1 Offline Point Location.

We begin with the offline point location problem in the asymmetric version in which the number of queries n is much larger than the size of the arrangement h. Formally, we have a set of n points in the plane together with a set of h, $h < n$, disjoint axis-aligned rectangles. The goal is to find the rectangle (if any) that contains each input point, all within $O(n \log\log_{n/h} n)$ time. Even though this is less popular than the general version of the problem (i.e., the case when $n = \Theta(h)$), the asymmetric case still arises from time to time in applications (e.g., see [10]). In fact, we will need this result for our four-dimensional maxima algorithm. As before, we assume $h \le n/\log^4 n$ since otherwise known results give us the desired bound.

Note that unlike some previous attempts [10], we cannot assume that the points are in sorted order. Another very significant difference between our variant and the general case considered by Chan et al. [10] relates to the "slab subproblem". In our case, the slab subproblem is trivial (i.e., point location among m horizontal lines) but all the known reductions from the general point location problem to the slab subproblem induce a $\log\log n$ factor penalty per query which is already too much for us.

To warm up, we begin with a very simple method. Assume $h \le \sqrt{n}/\log n$. In this case, we simply extend the horizontal and the vertical boundaries of each rectangle to form a grid with $O(h^2) = O(n/\log^2 n)$ cells. Each grid cell is either fully contained in a rectangle or lies outside all of them. With a simple sweep line algorithm, for each grid cell we can determine if it is contained in a rectangle, and if so, we store a pointer to that rectangle. This will use $O(h^2)$ space and will take $O(h^2 \log h)$ time to build. Fortunately, both these terms are sublinear. To solve the n point location queries, we use the linear time $\sqrt{n}$-way splitting algorithm of Theorem 1 on x and y-coordinates; for every point p, if we know the vertical slab that contains its x-coordinate and the horizontal slab that contains its y-coordinate, then we can calculate the grid cell that contains p. Finding the vertical (resp. horizontal) slab that contains each point is equivalent to h-way splitting with respect to *specific* splitters, namely, the x-coordinates (resp. the y-coordinate) of the slabs. This is known to be equivalent to the h-way splitter problem [15] and can be solved within the same time (i.e., $O(n \log\log_{n/h} n)$ which is in fact linear in this special case).

The above method crucially depends on the $O(h^2)$ space usage and in fact it is an expensive approach to turn the point location problem into a slab problem by paying a large space overhead. To make it useful, we need a preprocessing step which combines the grid based approach [2] with the first technique of Chan's point location strategy [9]. This recursive strategy uses an additional parameter, T, that intuitively captures the amount of space available to each subproblem.

Consider a subproblem for our preprocessing step which involves a set of m rectangles and a parameter $T > m$. The case when $m < \sqrt{T}$ is the base case: we employ the basic approach outlined above, by building an $m \times m$ grid (since we have a lot of space available). Otherwise, we build $\sqrt{T}$ vertical (resp. horizontal) slabs such that each slab contains $m/\sqrt{T}$ vertices (of the rectangles). This results in a grid of size T. Using a sweep line technique, we can determine for each grid cell whether:

(i) it is fully contained in a rectangle (we also store a pointer to the rectangle if this is the case), or

(ii) a rectangle horizontally cuts through it, or

(iii) a rectangle vertically cuts through it, or

(iv) none of the above (i.e., every rectangle intersecting the cell has a vertex inside the cell)

Because of the disjointness of the rectangles, (ii) and (iii) cannot happen simultaneously. Determining the above for each grid cell takes $O(m \log m + T \log T)$ time and storing it needs $O(T)$ space. Now consider a horizontal (resp. vertical) slab s and consider the rectangles that have a vertex inside it. We clip each such rectangle using s and then recurse inside s using parameter $\sqrt{T}$. Because we have assumed $T > m$, it follows that the recursion in fact terminates and will always reach a base case when $m < \sqrt{T}$.

Let $f(m, T)$ be the running time of the above procedure. We have,

$$f(m, T) = O(m \log m + T \log T) + 2\sqrt{T} f(m/\sqrt{T}, \sqrt{T}).$$

After i steps of recursion $f(m, T)$ is bounded by

$$O(2^i m \log m + 2^i T \log T) + 2^i T^{1-1/2^i} f(m/T^{1-1/2^i}, T^{1/2^i}).$$

The base case is when $m \le \sqrt{T}$ and for that we have $f(m, T) = O(m^2 \log m) = O(T \log T)$. This means, the recursion continues until $m/T^{1-1/2^i} \le T^{1/2^{i+1}}$ and thus, $i \le \log\log_{T/m} T - 1$. Combined with the base case, we can estimate $f(m, T) = O(T \log^2 T)$. This estimate is not tight but it suffices for now. To solve the offline point location, we will use $m = h$ and $T = n/\log^2 n$ which results in $O(n)$ construction time.

It remains to answer the n point location queries. We use the grid structures that we built above. At the root, we have n query points, $m = h$, and $T = n/\log^2 n$. Thus, at the root, we have a $\sqrt{n}/\log n \times \sqrt{n}/\log n$ grid. Handling the root is a bit easier than handling subproblem, so we first focus on the root. Very similar to the basic strategy, in $O(n)$ time, we can find the grid cell that contains each query point using the $\sqrt{n}$-way splitting algorithm. Consider a query point p inside a grid cell c. If there is a rectangle that completely contains c, then we can directly answer the query for p. If a rectangle horizontally (resp. vertically) cuts through c, then it follows from the disjointness of rectangles that no rectangle can vertically (resp. horizontally) cut through c which in return implies the answer to p can be found by recursing in the horizontal (resp. vertical) slab that contains c, and finally, if none of these is the case then we can recurse in either horizontal or vertical slab. Using the splitting algorithm, all the recursive problems can be built in $O(n)$ time.

Handling subproblems is only slightly more complicated and the case of the root outlined above. The main issue is that we don't have control on the number of points that get passed to each subproblem and thus we might be unable to apply the splitting algorithm. Consider a subproblem at depth i of the recursion. In this subproblem we will have $m' = \Theta(h/T^{1-1/2^i})$ rectangles, and a parameter $T' = T^{1/2^i}$. Assume we need to answer n' queries in this subproblem. We consider three cases.

1. $n' < T'$: Intuitively, in this case we have less than "expected" queries to answer to we simply use a classical method with $O((n'+m')\log n)$ time instead of recursing. Thus, in the remaining cases we will have $n' \geq T'$ which means the $\sqrt{T'}$-way splitting algorithm will run in linear time in the remaining cases.

2. $m' < \sqrt{T'}$ (the base case): Remember that in this case, during the preprocessing, we have built a $m' \times m'$ grid where each grid cell stores a pointer to the rectangles that contains it (if any). Since $n' \geq T$, using the m'-way splitting algorithm, we can find the grid cell that contains each query point and thus we can answer the queries in $O(n')$ time.

3. Otherwise: In this case, we have built a $\sqrt{T'} \times \sqrt{T'}$ grid. Using the $\sqrt{T'}$-way splitting algorithm, we find the grid cell that contains each query point and then recurse on the horizontal slab or the vertical slab similar to the root case.

The maximum depth of the recursion is at most $\log\log_{n/m} n \leq \log\log n$ and at the depth i of the recursion there are $2^i T^{1-1/2^i}$ subproblems, with each subproblem having a parameter $T' = T^{1/2^i}$. The total number of points (over all subproblems at depth i) involved in case 1 is at most $2^i T^{1-1/2^i} T^{1/2^i} = 2^i T = O(n/\log n)$. Similarly, the total number of rectangles involved in case 1 is $2^i m = O(h \log h)$. As there are only $O(\log\log n)$ recursions, the total running time (over all recursion depths) spent in case 1 is $O(n)$. The total running time spent in case 2 is also linear as it involves no recursions. This leaves step 3 only. The total time spent in step 3 is $O(nd)$ where d is the maximum recursion depth. Thus, we have the following.

Theorem 3. *A set of n offline point location queries on a set of h disjoint rectangles can be executed deterministically in $O(n \log\log_{n/h} n)$ time in a word RAM.*

4.2 Output-Sensitive Maxima in Four Dimensions.

To remind the reader, we denote the input set of n points with P. As before, we assume an upper bound h on the number of maxima points is known. Thus, we would like to find all the at most h maxima points among the n input points in four dimensions. We also assume $h \geq \sqrt{n}$ (if $h < \sqrt{n}$ we can still run our algorithm assuming $h = \sqrt{n}$). Let α_h be such that $h = n^{1-\alpha_h}$. Define $m = \sqrt{n^{\alpha_h}}$ and since $h \geq \sqrt{n}$ we have $m \leq \sqrt{n} \leq h$. Perform an m-way split along the fourth dimension, thus yielding sets $T_1, \dots, T_m$ in which each T_i contains $O(n/m)$ points and the fourth coordinate of any points in T_i is greater than those in sets $T_1, \dots, T_{i-1}$. The m-way split can be done in linear time.

We now use a sweep-based approach that curiously is not useful for the general maxima problem but it works quite well for the output-sensitive version. We begin from T_m and process the sets one by one under the invariant that before processing T_i, the maxima of the points in $T_{i+1}, \dots, T_m$ has already been computed and stored in a set M. To handle T_i, first, we prune any point of T_i that is dominated by a point in M. We call this the *prune step* and we shall present its details shortly. But notice that we can now recursively compute the maxima of the remaining points of T_i. Observe that the result of this recursion step can be added to M as a maxima point of T_i is a maxima point of P if it is not dominated by a point in M. We know $|T_i| = O(n/m)$, and $|M| \leq h$ and assuming the pruning step can be done in $P(n/m, h)$ time, the running time $T(n,h)$ of the algorithm will obey the following recursion:

$$T(n,h) = O(n) + mP(n/m,h) + \sum_{i=1}^{m} T(n/m, h_i)$$

where $\sum_{i=1}^{m} h_i = h$. In an upcoming lemma we will prove that $P(n, h) = O(n \log \log_{n/h} n)$. Thus,

$$
\begin{aligned}
&mP(n/m, h) = O\left(n \log \log_{n/(mh)}(n/m)\right) = \\
&O\left(n \log\left(\frac{(1-\alpha_h/2)\log n}{(1-\alpha_h/2)\log n - (1-\alpha_h)\log n}\right)\right) = \\
&O\left(n \log\left(\frac{1-\alpha_h/2}{\alpha_h/2}\right)\right) = O\left(n \log\log\frac{1}{\alpha_h}\right) = \\
&O\left(n \log\log_{n/h} n\right).
\end{aligned}
$$

Plugging this in the recursion for $T(n, h)$ we get that $T(n, h) = O(n \log \log_{n/h} n) + \sum_{i=1}^{m} P(n/m, h_i)$. Since the function $\log(\log(\frac{a}{a-\log(x)}))$ is a concave function of x, we have,

$$
\begin{aligned}
&\sum_{i=1}^{m} \log\left(\frac{\log n/m}{\log n/m - \log h_i}\right) \leq \\
&m \log\left(\frac{\log n/m}{\log n/m - \log(h/m)}\right).
\end{aligned}
$$

Our recursion thus solves to

$$T(n, h) = O(n(\log \log_{n/h} n) \log_{n/h} n).$$

We now look at the pruning step.

Lemma 5. *We have $P(n, h) = O(n \log \log_{n/h} n)$.*

Proof. Remember that the input to the pruning step is M and T_i. It is clear that we can ignore the fourth coordinate of all the points since all the points in M have their fourth coordinate larger than any points in T_i so in the rest of the proof we assume M and T_i are in the three dimensional space (or equivalently, we use M and T_i to refer to their projection onto the first three dimensions). Using a classical idea [19], the problem can in fact be turned to an offline point location problem: The region $\mathcal{R} \subset \mathbb{R}^3$ consisting of all the points dominated by points of M consists of vertices, and orthogonal faces. Intuitively, the idea is to look at $\mathcal{R}$ from "above" (or down the z-axis) which gives us a planar orthogonal arrangement, composed of $O(|h|)$ disjoint orthogonal rectangles and it can be built in $O(h \log \log h)$ time. By storing the z-coordinate of each face (or intuitively, it's "elevation"), once we find the face of the arrangement that contains the projection of a point $p \in T_i$, we can determined if p is dominated by a point in M or not. Thus, lemma follows from Theorem 3.
□ □

In the next section, we show how to generalize this idea to higher dimensions.

4.3 Output-Sensitive Maxima in Higher Dimensions.

The general strategy to compute the maxima in higher dimensions is not too complicated and in fact it is very similar to the four-dimensional case. We split the points according to the value of d-th dimension thus dividing the points into "slabs" perpendicular to that dimension. We follow a sweep approach to prune some of the points and then recurse on the points within each slab. During the sweep, we will encounter a different problem and in one dimension lower.

As before, we assume $h \geq \sqrt{n}$, let α_h be such that $h = n^{1-\alpha_h}$, and $m = \sqrt{n^{\alpha_h}}$. Perform an m-way split along the d-th dimension, thus yielding sets $T_1, \dots, T_m$. The m-way split can be done in linear time as before. The general layout of the algorithm is the same as the one for the four dimensions and the only difference is the pruning step. If we denote the time it takes to compute the maxima of n points in d dimensions with $T_d(n, h)$, we have

$$T_d(n, h) = O(n) + mP_{d-1}(n/m, h) + \sum_{i=1}^{m} T_d(n/m, h_i) \qquad (1)$$

where $P_{d-1}(n/h, h)$ is the time it takes to process a given set of n/h points in $(d-1)$-dimensional space and remove the points dominated by at least a point from another set of h points.

The main point of departure from the $d = 4$ case is the following lemma. Unfortunately, for dimensions five and above, we no longer have access to the nice reduction to the point location problem in two dimensions lower so we must try a different strategy.

Lemma 6. *We have $P_d(n, h) = O(n(\log\log_{n/h} n)(\log_{n/h} n)^{d-3})$.*

Proof. Consider two sets of points P and M in d-dimensional space with $|P| = n$ and $|M| = h$. To prove the lemma we must show that we can prune any point of P that is dominated by a point of M within the time bound in the statement of the lemma. We use induction and we assume the lemma holds in $(d-1)$-dimensional space. Clearly Lemma 5 is the base case.

Let $t = \sqrt{n/h}$. We perform a t-way split on the d-th dimension on set P, resulting in sets $P_1, \dots, P_t$, arranged according to the decreasing value of the d-th coordinate (i.e., points in P_i have larger d-th coordinate than points in P_{i-1}). This gives us t slabs perpendicular to the d-th dimension such that P_i is the set of points contained in the i-th slab. Distribute the points of M into these slabs as well. This phase takes $O(n + h\log\log_{h/t} h)) = O(n)$ time. Let $M_1, \dots, M_t$ be the resulting sets, arranged according to the decreasing order of the d-th coordinate.

For every $i = 1, \dots, t$, using a $(d-1)$-dimensional pruning subroutine, delete the points in P_i that is dominated by a point in $U_i = \cup_{j=1}^{i-1} M_j$. Finally, recursively prune P_i by M_i. Since $|P_i| = O(n/t)$ and $|U_i| \le h$, we get the following recursion:

$$P_d(n,h) = O(n) + tP_{d-1}(n/t, h) + \sum_{i=1}^{t} P_d(n/t, h_i)$$

$$\text{where } \sum_{i=1}^{t} h_i = h.$$

Now observe that

$$\frac{\log(n/t)}{\log(n/t) - \log h} \le \frac{\log n}{\log n - \log h - \log t} \le \frac{2\log n}{\log n - \log h},$$

which means

$$\begin{aligned} & tP_{d-1}(n/t, h) = \\ & O\left(n \log\left(\log_{n/(th)}(n/t)\right)\left(\log_{n/(th)}(n/t)\right)^{d-4}\right) \\ & = O\left(n(\log\log_{n/h} n)\left(\log_{n/h} n\right)^{d-4}\right). \end{aligned}$$

The rest of the proof is very similar to the proof of Lemma 5: by a similar concavity argument, we can show that the maximum value of the recursion is obtained when $h_i = h/t$. The lemma follow by observing that the depth of the recursion is $\log_t n = O(\log_{n/h} n)$. □ □

By plugging in the value for $P_d(n, h)$ in (1) we obtain the following theorem.

Theorem 4. *The maxima of a set of n points with integer coordinates in d-dimensional space can be computed deterministically in $O(n(\log\log_{n/h} n)(\log_{n/h} n)^{d-3})$ time using a word RAM, in which h is the number of maxima points.*

5 Conclusions and Open Problems

In this paper, we gave very fast algorithms for computing output-sensitive maxima in three and higher dimensions. We also justified our belief that these algorithms are difficult to improve.

This work leaves many interesting problems to be tackled. We mention the two most interesting ones: First, is it possible to answer n offline point location queries on a non-orthogonal arrangement of size h in linear time, assuming $h \leq n^{\varepsilon}$, for some constant ε? We note that an affirmative answer to this question will automatically improve the offline point location results of Chan and Pătraşcu [11]. Second, is it possible to find the maxima of a set of points in three dimensions in $o(n \log\log n)$ expected time? Alternatively, one can also formulate this as a point location problem: is it possible to answer n offline points locations queries in an arrangement of size n in $o(n \log\log n)$ expected time? We believe the answer to all these questions should be positive even though the current techniques seem insufficient to tackle them.

References

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *Proc. 50th IEEE Symposium on Foundations of Computer Science*, pages 129–138, 2009.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[3] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th IEEE Symposium on Foundations of Computer Science*, pages 534–, 1998.

[4] J. Bentley, K. Clarkson, and D. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9(2):168–183, 1993.

[5] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[6] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proc. 1st ACM/SIAM Symposium on Discrete Algorithms*, pages 179–187, 1990.

[7] J. L. Bentley and M. I. Shamos. Divide and conquer for linear expected time. *Information Processing Letters*, 7(2):87–91, 1978.

[8] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16(4):361–368, 1996.

[9] T. M. Chan. Persistent predecessor search and orthogonal point location in the word RAM. In *Proc. 22nd ACM/SIAM Symposium on Discrete Algorithms*, 2011. To appear.

[10] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry*, pages 1–10, 2011.

[11] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, II: offline search. *ACM Transactions on Algorithms*, submitted, 2010. Preliminary version in STOC'07.

[12] K. L. Clarkson. More output-sensitive geometric algorithms. In *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pages 695–702, 1994.

[13] K. L. Clarkson, W. Mulzer, and C. Seshadhri. Self-improving algorithms for coordinate-wise maxima. In *Proc. 28th ACM Symposium on Computational Geometry*, pages 277–286, 2012.

[14] M. J. Golin. A provably fast linear-expected-time maxima-finding algorithm. *Algorithmica*, 11:501–524, 1994.

[15] Y. Han and M. 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*, pages 135–144, 2002.

[16] S. Kapoor and P. Ramanan. Lower bounds for maximal and convex layers problems. *Algorithmica*, 4(1-4):447–459, 1989.

[17] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proc. 1st ACM Symposium on Computational Geometry*, pages 89–96, 1985.

[18] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22:469–476, 1975.

[19] C. Makris and A. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, 1998.

[20] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proc. 33rd International Conference on Very Large Databases*, pages 15–26, 2007.

[21] S. Sen and N. Gupta. Distribution-sensitive algorithms. *Nordic Journal of Computing*, 6(2):194–211, 1999.