Journal of Discrete Algorithms 30 (2015) 78–95

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Reporting and counting maximal points in a query orthogonal rectangle [☆]

Ananda Swarup Das [a,*], Prosenjit Gupta [b], Kishore Kothapalli [c], Kannan Srinathan [c]

[a] IBM India Research Labs, New Delhi, India
[b] Heritage Institute of Technology, Kolkata, India
[c] International Institute of Information Technology, Hyderabad, India

ARTICLE INFO

Article history:
Received 6 May 2013
Received in revised form 4 December 2014
Accepted 4 December 2014
Available online 9 December 2014

Keywords:
Computational geometry
Range queries
Maximal points

ABSTRACT

In this work we show that given a set S of n points with coordinates on an $n \times n$ grid, we can construct data structures for (i) reporting and (ii) counting the maximal points in an axes-parallel query rectangle in sub-logarithmic time. We assume our model of computation to be the word RAM with size of each word being $\log n$ bits.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

A point $p = (p(x), p(y)) \in \mathbb{R}^2$ is dominated by another point $q = (q(x), q(y)) \in \mathbb{R}^2$ if $p(x) < q(x)$ and $p(y) < q(y)$. Given a set S of n points in $\mathbb{R}^2$, a point $p \in S$ is maximal if it is not dominated by any point $q \in S$. Similarly, a point q is

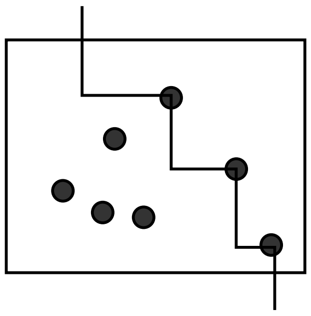

Fig. 1. The points in the staircase are the maximal points.

☆ Preliminary versions of this paper containing solutions to the reporting and counting problems appear in Proceedings of WALCOM 2012 [6] and WALCOM 2013 [7].

* Corresponding author.

E-mail addresses: anandaswarup@gmail.com (A.S. Das), prosenjit_gupta@acm.org (P. Gupta), kkishore@mail.iiit.ac.in (K. Kothapalli), srinathan@mail.iiit.ac.in (K. Srinathan).

http://dx.doi.org/10.1016/j.jda.2014.12.002
1570-8667/© 2014 Elsevier B.V. All rights reserved.

Table 1
Different results known for the reporting problem.

Query time	Storage space	Model	Updates	Source
$O(\log^2 n + k)$	$O(n \log n)$	Pointer	Dynamic	[3]
$O(\log n + k)$	$O(n \log n)$	Pointer	Static	[11]
$O(\log n + k)$	$O(n \log n)$	Pointer	Static	[10]
$O(\frac{\log n}{\log\log n} + k)$	$O(n \frac{\log n}{\log\log n})$	**word RAM**	**Static**	**this work**
$O((k+1)\log\log n)$	$O(n \log^{\varepsilon} n)$	word RAM	Static	[14]
$O(\frac{\log n}{\log\log n} + k)$	$O(n \log^{\varepsilon} n)$	word RAM	Static	[2]
$O((k+1)(\log\log n)^2)$	$O(n \log\log n)$	word RAM	Static	[14]
$O(\frac{\log n}{\log\log n} + k \log\log n)$	$O(n \log\log n)$	word RAM	Static	[2]
$O(n)$	$O((k+1)\log^{\varepsilon} n)$	word RAM	Static	[14]

a minimal point if it does not dominate any point in S. See Fig. 1. Given the set S, the problem of finding maximal (respectively minimal) points for the set S is fairly well studied and can be solved in $O(n \log n)$ time using $O(n)$ space (see [15]). The problem of finding maximal points is widely studied in the database community where the maximal points are better known as *skyline points*. The skyline points are helpful in decision making tools where the number of points to consider is too big [4]. The skyline query generally returns the *interesting points* which are hopefully not too many.

In this work, we propose data structures to (i) report and (ii) count maximal points in an orthogonal query rectangle.

The first problem is important as it is believed that a user is often not concerned about the entire data set and is interested to find a result within his/her query window of interest. Of course, one can propose an algorithm where given a query window, every point in the data set is checked to find if it fits in the window and then the maximal points for the resulting subset are returned as an answer. But, such an approach may be too expensive in terms of query time when the number of points in the data set is very large. Brodal et al. in [3] proposed the first dynamic solution in a pointer machine model. Their solution uses a data structure of size $O(n \log n)$ and can be queried to report maximal points in $O(\log^2 n + k)$ time, where k is the size of the output. In this work, we consider the static case of the problem in the word RAM model of computation with size of each word being $\theta(\log n)$ and propose a data structure of size $O(n \frac{\log n}{\log\log n})$ that can be queried in $O(\frac{\log n}{\log\log n} + k)$ time.

As stated above, the maximal points are used in decision making tools to reduce the size of the result. Thus, counting maximal points may be useful to find the size of the resulting maximal points in a query window especially if the result has to be transferred using network bandwidth. For the problem, we propose a data structure of size $O(n \frac{\log^3 n}{\log\log n})$ that can be queried in $O(\frac{\log n}{\log\log n})$ time.

1.1. Our contributions

In this work, the first problem that we study is the following.

Problem 1. Given a set R of n points on an $n \times n$ grid, that is $R = \{(x_i, y_i) | x_i \in \{1, n\}, y_i \in \{1, n\}\}$, preprocess R into a data structure such that given an orthogonal query rectangle q, the maximal points in $q \cap R$ can be reported efficiently.

For the problem, we have the following result.

Theorem 1. *A set R of n points on an $n \times n$ grid, that is $R = \{(x_i, y_i) | x_i \in \{1, n\}, y_i \in \{1, n\}\}$, can be preprocessed into a data structure of size $O(n \frac{\log n}{\log\log n})$ such that given a query rectangle q, the maximal points in $q \cap R$ can be reported in $O(\frac{\log n}{\log\log n} + k)$ time.*

Table 1 summarizes the different output sensitive results known in the literature for the problem.

Special note. Recently, Kejlberg-Rasmussen et al. in [12] have studied the problem in the external memory model and have presented an algorithm to report maximal points in a query rectangle in $O((\frac{n}{B})^{\varepsilon} + \frac{k}{B})$ I/Os, where k is the number of points to be reported and B is the block size.

The second problem of this work is the following.

Problem 2. Given a set R of n points on an $n \times n$ grid, that is $R = \{(x_i, y_i) | x_i \in \{1, n\}, y_i \in \{1, n\}\}$, preprocess R into a data structure such that given an orthogonal query rectangle q the count of the maximal points in $q \cap R$ can be reported efficiently.

For the problem, we have the following result.

Table 2
Different results for the counting version.

Query time	Storage space	Model	Updates	Source
$O(\log n)$	$O(n \log n)$	Pointer	Static	[10]
$O(\frac{\log n}{\log\log n})$	$O(n\frac{\log^3 n}{\log\log n})$	word RAM	Static	**this work**
$O(\frac{\log n}{\log\log n})$	$O(n)$	word RAM	Static	[2]

Theorem 2. *Given a set R of n points on an $n \times n$ integer grid, we can preprocess the points into a data structure of size $O(n\frac{\log^3 n}{\log\log n})$ words, such that given an axis parallel query rectangle $q = [a, b] \times [c, d]$, we can count the number of maximal points in $R \cap q$ in $O(\frac{\log n}{\log\log n})$ time.*

Table 2 summarizes the different results known in the literature for the problem.

Special note. Recently, Brodal et al. in [2] studied the problem of counting maximal points in a query rectangle and correlated the problem with finding reachability in butterfly networks. In this work, we solve the problem by first proposing an efficient solution for counting maximal points in a restricted grid of size $[1, \log^\rho n] \times [1, n] : 0 < \rho \le \frac{1}{2}$ where we actually precompute the results and then find the count of maximal points in a query rectangle from the precomputed results. We then extend this solution to the general setting. Although the proposed counting structure is not optimal, it serves the purpose of showing that reporting and counting can be solved under the same framework.

Assumption. Throughout the work, we assume that each point in R has distinct x- and y-coordinates.

2. Outline of the paper

(1) We first present a preliminary data structure with $O(n \log n)$ space and $O(\log n + k)$ query time, that in essence shows how to use 1-dimensional "range maximum queries" over a 2-dimensional range tree (with fractional cascading pointers) in order to support "2-dimensional range maximal points" reporting queries. Both range maximum queries and 2-dimensional range trees with fractional cascading are discussed in the next section.

(2) Next, we show that if the degree of the range tree is increased to roughly $O(\sqrt{\log n})$, then, an order of $O(\log\log n)$ improvement is possible in both space and query time, given that a subproblem of restricted size can be solved in a node in $O(1)$ time. We tackle this issue by proposing an algorithm that identifies the nodes with useful output and by utilizing fast data structures for "restricted 3-sided range successor queries" and "rank/select queries" (see [17]) to support the query on these nodes.

(3) We finally extend the data structure to support "2-dimensional range maximal points" counting queries.

3. Preliminaries

Range tree with fractional cascading: Consider the following problem, *"Given a set S of n points with distinct x- and y-coordinates in $\mathbb{R}^2$, preprocess S such that given a query rectangle q, we can efficiently report the points in $S \cap q$"*. A simple way to solve the problem is to use a range tree which is constructed as follows:

1. Build a primary tree T_x which is a balanced binary search tree with all the leaves at the same level. The leaves store the x-coordinates of the points in S in non-decreasing order of their values.
2. Each internal node $\mu \in T_x$ is assigned an interval $int(\mu)$ which is union of the discrete intervals associated with the leaf nodes of the subtree rooted at μ.
3. Each internal node μ also maintains an auxiliary array A_μ which stores the y-coordinates of the points present in the subtree rooted at μ. These y-coordinates are sorted in non-decreasing order of their values.
4. For any node $\phi \in T_x$, it must be noted that $int(\phi) = int(u) \cup int(w)$ where u, w are respectively the left and the right children for ϕ. Also $A_\phi = A_u \cup A_w$. Thus, with each element $y \in A_\phi$, we store two pointers pointing to the smallest values greater than y in A_u and A_w respectively. The idea of maintaining such pointers in known as *fractional cascading*.

In the subsequent sections, we will refer to this tree as a *two layer range tree*.

Lemma 1. *The storage space needed by the range tree is $O(n \log n)$.*

Proof. The total space needed by the primary tree T_x with n leaves is $O(n)$. Next, it should be noted that for any particular point $p = (p(x), p(y)) \in S$, its y-coordinate can be present in at most one auxiliary array A_ϕ among all the auxiliary arrays associated with the internal nodes of the tree T_x at a particular depth d. Thus, if we add the sizes of all the auxiliary arrays

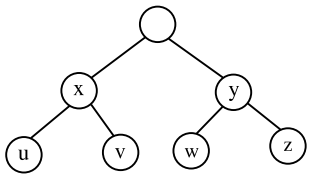

Fig. 2. The tree with its nodes.

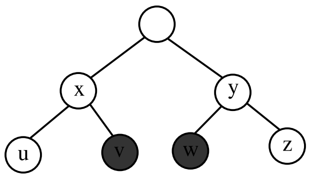

Fig. 3. The segment $[a, b]$ is allocated to the nodes colored black.

at the depth d of the tree T_x, it is equal to $O(n)$. As T_x is a height balanced binary search tree with n elements, its height is bounded by $O(\log n)$. Hence, the total storage needed by the range tree is $O(n \log n)$. □

Query algorithm: Given a query rectangle $[a, b] \times [c, d]$, we do the following:

1. The segment $[a, b]$ is allocated to a node $\mu \in T_x$, if $int(\mu) \subset [a, b]$ but $int(parent(\mu)) \not\subseteq [a, b]$. The node μ is known as a *canonical node*. Let S_{can} be the set of such nodes. Note that S_{can} is computed every time for the each query separately.
2. Search the auxiliary array A_{root} and find the smallest element $y(1) \ge c$ and the largest element $y(2) \le d$. Then, follow the pointers for the elements $y(1)$ and $y(2)$ and find the smallest element $y(1)' \ge c$ and the largest element $y(2)' \le d$ for each of the auxiliary arrays $A_{\phi(l)} : \phi(l) \in S_{can}$.
3. Report all the points whose y-coordinates are in the range of $[y(1)', y(2)']$ for the array $A_\phi(l) : \phi(l) \in S_{can}$.

Lemma 2. *There are $O(\log n)$ nodes in S_{can}.*

Proof. See Fig. 2. Let x be the parent for the nodes u, v and y be the parent for w. Also, let x, y be siblings. We know that $int(x) = int(u) \cup int(v)$. Suppose $int(u) \subset [a, b]$ and $int(v) \subset [a, b]$. Then, $int(x) \subseteq [a, b]$. Thus, the segment $[a, b]$ will be allocated to the node x or any of its ancestors but not to the nodes u, v. This actually means that no two siblings of the tree can be allocated the segment $[a, b]$. Also notice that if $int(u) \subset [a, b]$ but $int(v)$ is not contained in $[a, b]$, then $int(w)$ is not contained in $[a, b]$. Thus, this point ensures that if the nodes of the tree are arranged in level-wise order from left to right for a particular level and if a node to which $[a, b]$ is allocated is a left child of its parent, then any succeeding node in the level wise order of nodes at that level cannot be allocated the segment $[a, b]$. Hence, the only possible way the segment $[a, b]$ can be allocated to two nodes at the same level is as shown in Fig. 3. Thus, the maximum number of nodes to which the segment $[a, b]$ can be allocated at a particular level is two. Since the height of a *height balanced binary search tree* with n leaf nodes is $O(\log n)$. Thus, the number of canonical nodes to which the segment $[a, b]$ can be allocated is $O(\log n)$. □

Range maximum data structure: The problem for range maximum can be defined as follows: *"Given an array A of n integers, preprocess A such that given two indices (i, j) for the array A, we can return the index $t : i \le t \le j$ storing the largest value in $A[i \dots j]$."*

A simple solution for the problem is as follows: For each index i in the array A and for every choice of r for $0 \le r \le \log n$, we form two intervals $[i, i + 2^r - 1]$ and $[i - 2^r + 1, i]$. For these two intervals, we compute the indices with the maximum values in $A[i, i + 2^r - 1]$ and $A[i - 2^r + 1, i]$. We denote these two indices as t_1 and t_2 respectively. For the element in $A[i]$, we maintain two arrays, each of size $O(\log n)$ and are denoted by $B^+_{A[i]}$ and $B^-_{A[i]}$. We store the index t_1 at the rth index of $B^+_{A[i]}$. Similarly, the index t_2 is stored at the rth index of $B^-_{A[i]}$. Given two query indices (i, j) we compute $r' = \lfloor \log_2(j - i) \rfloor$ and find the two indices (t_1, t_2) in $B_{A[i]^+}[r']$ and $B_{A[j]^-}[r']$. We compare the two values $A[t_1]$ and $A[t_2]$ and return the index storing the maximum value. Clearly, the range maximum queries can be answered in $O(1)$ time using $O(n \log n)$ storage space. In the subsequent sections, we denote the range maximum query on $A[i, j]$ as $RMQ(A[i, j])$. Though the technique discussed here is query efficient, it is not very storage efficient. However, the following result is known for the problem from [8].

Theorem 3. *(See [8].) For an array of n elements from a totally ordered set, there exists an algorithm for RMQ problem with query time complexity $O(1)$ and bit space complexity $2n + o(n)$ bits.*

Recently in [1], Brodal et al. studied the problem of range maximum/minimum for 2-d array A of size $m \times n$ and proposed a beautiful linear space data structure that can be preprocessed in linear time and can be queried with a rectangle $q = [i_1, \dots, i_2] \times [j_1, \dots, j_2]$ to report the position of the minimum element in a rectangle range within A in $O(1)$ time. It should be noted that the range maximum query for 2-d array is just not a simple extension of its 1-d counterpart as all the memory indices accessed in the 2-d array are not actually contiguous unlike its 1-d counterpart.

The restricted range successor problem: Consider the following problem: *"Given a set S of n points from an integer grid of $[1, \sqrt{\log n}] \times [1, n]$ where every point has a distinct y-coordinate, preprocess S into a data structure such that given a query rectangle $[a, b] \times [c, d]$ where $(a, b) \in [1, \sqrt{\log n}] \times [1, \sqrt{\log n}]$ and $(c, d) \in [1, n] \times [1, n]$, we can report the minimum x-coordinate in $O(1)$ time."*

A simple solution for the problem is as follows:

Preprocessing:

1. Store the y-coordinates of the points in S in an array A. Sort the array A in non-decreasing order of its values.
2. Construct a height balanced binary search tree T_y, the leaves of which store the y-coordinates of S in non-decreasing order. To each internal node of the tree $\mu \in T_y$, store a value denoted as *key* which is equal to the median of the values present in the subtree rooted at μ.
3. For each possible pair of points $(i_1, i_2) \in [1, \sqrt{\log n}] \times [1, \sqrt{\log n}]$, form an interval $[i_1, i_2]$ and do the following:
   (a) For each $y \in A$, form two 3-sided anchored rectangles of the form $q_1 = [i_1, i_2] \times (-\infty, y]$ and $q_2 = [i_1, i_2] \times [y, \infty)$.
   (b) For each such rectangles q_1 (respectively q_2) maintain an array of size $O(\log n)$ and denote the array as $pathrecord_{q_1}$ (respectively $pathrecord_{q_2}$). Initially all the values for the array $pathrecord_{q_1}$ and $pathrecord_{q_2}$ are set to *zero*.
   (c) Visit each ancestor μ for the leaf storing the value y and do the following:
      i. Consider the value *key* at the node μ. If $key \le y$, form a rectangle $[i_1, i_2] \times [key, y]$. Else, form a rectangle $[i_1, i_2] \times [y, key]$.
      ii. Let the node μ be at a depth h in T_y. Find the point p with minimum x-coordinate in the rectangle. If $key \le y$, store in $pathrecord_{q_1}[h]$ the value of the minimum x. Else if $key > y$, store in $pathrecord_{q_2}[h]$ the value of the minimum x.
4. Also, maintain the data structure of [9] to find the least common ancestor for two given leaf nodes of the tree T_y in $O(1)$ time.

Query algorithm: Given a query rectangle $[a, b] \times [c, d] : (a, b) \in [1, \sqrt{\log n}] \times [1, \sqrt{\log n}]$, $(c, d) \in [1, n] \times [1, n]$, we do the following:

1. Find the least common ancestor for the values c, d in the tree T_y using the data structure of [9]. Notice that there are n points in the set S and each point has a distinct y-coordinate. Also, $(c, d) \in [1, n] \times [1, n]$. Thus, the values c, d are present in the cth and the dth leaf nodes for the tree T_y assuming the leftmost leaf node as the 1st leaf node.
2. Let the least common ancestor be at depth h and let the *key* stored at the least common ancestor be y. We form two rectangles namely $q_1 = [a, b] \times [c, \infty)$ and $q_2 = [a, b] \times (-\infty, d]$.
3. We compare the values at $pathrecord_{q_1}[h]$ and $pathrecord_{q_2}[h]$ and return the one which is minimum.

The above data structure needs $O(n \log^2 n)$ storage space and can be queried to find the point with minimum x coordinate in the query rectangle in $O(1)$ time. However, the following result is known from [17]:

Theorem 4. *(See [17].) If the coordinates of the points are on an integer grid of $[1, \frac{\log n}{\log\log n}] \times [1, n]$, then there exists a linear space data structure that can be queried in $O(1)$ time to report the point with smallest x-coordinate in $[a, \infty) \times [c, d]$ where $a \in [1, \frac{\log n}{\log\log n}] \times [1, \frac{\log n}{\log\log n}]$ and $(c, d) \in [1, n] \times [1, n]$.*

To report the point with smallest x-coordinate in $[a, b] \times [c, d]$, we can simply find the point with smallest x coordinate in $[a, \infty) \times [c, d]$. If the x-coordinate for the reported point is less than or equal to b, we return the point, else we return *null*.

Rank and select queries:

Rank query: Let S be a character string of length n over a finite ordered alphabet $\sigma = \{1, 2, \dots, |\sigma|\}$. For any character $c \in \sigma$ and any position i, a rank query denoted by $rank_c(S, i)$ reports the number of c in S from position 1 to position i. Consider the example as given in [17]. Let $S = 231131321$. The query $rank_3(S, 6) = 2$.

Select query: A select query $select_c(S, j)$ returns the position of the jth occurrence of the character c in S. The query $select_3(S, 2) = 5$.

For the rank query and the select query, the following result is known from [5].

Lemma 3. *(See [5].) A binary string $S(1, n)$ can be represented using $n + o(n)$ bits of space while supporting rank query as well as select query in $O(1)$ time.*

4. A preliminary solution for Problem 1

Preprocessing:

1. We construct the two layer range search tree T_x as discussed in the previous section. However, each auxiliary array A_ϕ associated with the internal node $\phi \in T_x$, stores the y-coordinates of the points present in the leaves of the subtree rooted at μ in non-increasing order.
2. Additionally, at each internal node $\phi \in T_x$, we maintain an auxiliary array B_ϕ. The ith node of B_ϕ stores the x-coordinate $p(x)$ of the point $p = (p(x), p(y))$ whose y-coordinate $p(y)$ is stored at $A_\phi[i]$.
3. Preprocess B_ϕ into an instance of the data structure of Theorem 3. Remember that from [8], we know that an $RMQ(B_\phi[i, j])$ can be performed in $O(1)$ time. An instance of the data structure of [8] for performing $RMQ(B_\phi[i, j])$ needs $O(|B_\phi|)$ bits.
4. Also, for each element $y \in A_\phi$ we maintain a pointer to its position in $A_{parent(\phi)}$. Thus, each element $y \in A_\phi$ has now three pointers.

Lemma 4. *The above data structure needs $O(n \log n)$ space.*

The proof for the above lemma is similar to the proof of Lemma 1.

Query algorithm:

1. Given a query rectangle $q = [a, b] \times [c, d]$, we assign the segment $[a, b]$ to $O(\log n)$ canonical nodes of the tree T_x.
2. Let S_{can} be the set of such $O(\log n)$ canonical nodes. Arrange these nodes in order of their occurrences from right to left in the tree T_x.
3. Next, search the array A_{root}, the auxiliary array which stores the y-coordinates of the points present in the tree rooted at *root* and find the largest value $y_1 \leq d$. Then, using fractional cascading, find the indices i in the auxiliary arrays $A_{\phi(l)}$ associated with each of the canonical nodes $\phi(l) \in S_{can}$ such that $A_{\phi(l)}[i]$ is the largest value smaller than d in $A_{\phi(l)}$.
4. We begin our search from the rightmost canonical node $\phi(1)$ in S_{can}.
5. Search the array $A_{\phi(1)}$ and find the index j storing the smallest value $\geq c$.
6. Run $RMQ(B_{\phi(1)}[i, j])$ (using the technique of [8]) and find the index $t : i \leq t \leq j$ with the largest x-coordinate in $B_{\phi(1)}[i, j]$.
7. Next, run $RMQ(B_{\phi(1)}[i, t - 1])$. Repeat RMQ queries until the value in $B_{\phi(1)}[i]$ is reported. Notice that the point $p = (p(x), p(y))$ whose x coordinate is stored at $B_{\phi(1)}[i]$ will be reported as the last maximal point from the node $\phi(1)$. (The proof for the same follows in the next section.)
8. Move to the next rightmost node $\phi(2) \in S_{can}$ and consider only the points with y-coordinates that are greater than $p(y)$, the y-coordinate of the last maximal point reported from the node $\phi(1)$ and are in the range $[c, d]$.
9. By step 3, we already have the index i for the array $A_{\phi(2)}$ such that $A_{\phi(2)}[i]$ is the largest value smaller than d. By following the pointer for the value $p(y) \in A_{\phi(1)}$ to its corresponding position in $A_{parent(\phi(1))}$ and then chasing the subsequent pointers, we can find the index j for the smallest value greater than $p(y)$ in $A_{\phi(2)}$ (the details follow in the proof for Lemma 6).
10. Repeat steps 6 and 7 at the node $\phi(2)$.
11. Continue the above steps until all the nodes in S_{can} have been visited.

Lemma 5. *The point $p = (p(x), p(y))$ whose x-coordinate is stored in $B_{\phi(1)}[i]$ and is reported as the last point from the node $\phi(1)$ (by step 7) is a maximal point.*

Proof. For any node $\phi(2)$ which is to the left of $\phi(1)$, the leaves present in the subtree rooted at $\phi(2)$ store values smaller than the smallest value stored in the leftmost leaf of the subtree rooted at $\phi(1)$. Thus, no point from any other subtree rooted at any other canonical node in S_{can} can dominate p. Next, the auxiliary array $A_{\phi(1)}$ is sorted in non-increasing order of the y-coordinates stored. Hence $A_{\phi(1)}[i] > A_{\phi(1)}[i + 1] > \ldots > A_{\phi(1)}[j]$. Hence the claim holds. $\square$

Lemma 6. *The smallest y-coordinate greater than $p(y)$ in $A_{\phi(2)}$ can be found using fractional cascading.*

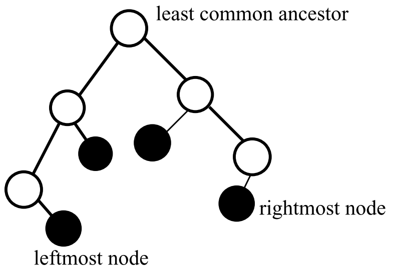

Fig. 4. The segment $[a, b]$ is allocated to the nodes colored black.

Proof. Notice that the node $\phi(2)$ has to be to the left of $\phi(1)$ and it cannot be a descendant of $\phi(1)$. This is because, if $\phi(2)$ is a descendant of $\phi(1)$, then it means $int(\phi(2)) \subset [a, b]$ and $int(\phi(1)) \subset [a, b]$ and thus the segment $[a, b]$ cannot be allocated to $\phi(2)$. Thus, $\phi(2)$ has to be either a left child of some ancestor for $\phi(1)$ or it has to be a right child for some ancestor of the leftmost node in S_{can} (assuming, it is not the leftmost node itself). See Fig. 4. If $\phi(2)$ is a left child for some ancestor of $\phi(1)$, then we can start following the pointer for the value $p(y) \in A_{\phi(1)}[i]$ to its corresponding position in the array $A_{parent(\phi(1))}$ and continue to do so until we reach the parent of $\phi(2)$ from where we find the largest value smaller than y_1 in $A_{\phi(2)}$. Remember for each value y in an auxiliary array A_ϕ we maintain two pointers pointing to the smallest values greater than y in the auxiliary arrays associated with the left and the right children of ϕ. If $\phi(2)$ is a right child for some ancestor of the leftmost node, we can follow similar steps. That is, we move to the least common ancestor for the rightmost and the leftmost nodes and then start descending the path from the least common ancestor to the leftmost node. □

Lemma 7. *The query algorithm reports all the maximal points in the query rectangle.*

Proof. For any node $\phi(l)$, if it is the rightmost node, we start reporting the maximal points by finding the point $p = (p(x), p(y))$ with the maximum x-coordinate. This point cannot be dominated by any other point as this is the rightmost point in the query rectangle. The next point that is reported is the rightmost point among all the points that are above $p(y)$ and are in the query rectangle. We continue to do so as long as there are maximal points from the node $\phi(l)$. However, if $\phi(l)$ is not the rightmost node, we consider the last point $p = (p(x), p(y))$ reported from the node $\phi(l-1)$. Notice that this point p is the topmost and even leftmost maximal point reported as of yet. By step 3, we have the index i for the array $A_{\phi(l)}$ such that $A_{\phi(l)}[i]$ is the largest value smaller than d. By step 8, we find the index j such that $A_{\phi(l)}[j]$ is the smallest value greater than $p(y)$. We repeat *RMQ* at the node $\phi(2)$ and report the point p' with the maximum x coordinate. The point p' has the maximum x coordinate above $p(y)$ and hence cannot be dominated by any other point to the left of p in the query rectangle. Also, this point cannot be dominated by any other point to the right of p as their y-coordinates are less than the y-coordinate of p'. We repeat similar steps until all the canonical nodes in S_{can} are visited. Hence the claim holds. □

Lemma 8. *The query algorithm reports all the maximal points in the rectangle in $O(\log n + k)$ time.*

The proof for the above lemma is similar to the proof of Lemma 24 which we discuss later. Hence we skip the proof for the above lemma.

The above technique was used in [11] for reporting maximal points. A way to achieve the sub-logarithmic query time is to reduce the height of the tree by increasing the degree of each internal node of the tree to $O(\sqrt{\log n})$. Thus, the height of the tree is reduced to $O(\frac{\log n}{\log\log n})$.

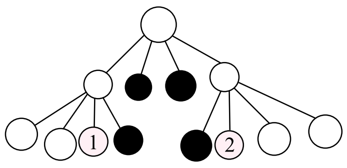

Fig. 5. The segment $[a, b]$ is allocated to the nodes colored black and the nodes marked 1, 2. However, only the nodes marked 1, 2 have maximal points.

Given a query rectangle $[a, b] \times [c, d]$, if we allocate the segment $[a, b]$ in a way similar to the step 1 of the query algorithm for reporting points in a rectangle, the number of our canonical nodes increases to $O(\frac{\log^{3/2} n}{\log\log n})$. The challenge is to wisely identify the nodes holding points present in the maximal chain in the query rectangle. We will call such canonical nodes as *profitable nodes* in the rest of the text. It should be noted that while targeting a sub-logarithmic query time bound, it is crucial to quickly (preferably in $O(1)$ time) identify if a canonical node is a *profitable node*. The reason is as follows. In the above discussed algorithm, the height of the tree is $O(\log n)$ and so is the number of the canonical nodes. Thus, even if a node does not hold any maximal point for the query rectangle and we accidentally visit the node, we still manage to get a query time of $O(\log n + k)$. However, this is not the case when the height of the tree is reduced to $O(\frac{\log n}{\log\log n})$ as the number of canonical node increases to $O(\frac{\log^{3/2} n}{\log\log n})$. In case all the canonical nodes are profitable nodes and we visit all these canonical nodes our query time will be $O(\frac{\log n}{\log\log n} + \frac{\log^{3/2} n}{\log\log n} + k)$. As $k \geq \frac{\log^{3/2} n}{\log\log n}$, our query time becomes $O(\frac{\log n}{\log\log n} + k)$. However, if $k < \frac{\log^{3/2} n}{\log\log n}$, then we do not have the luxury to visit all the canonical nodes as we are targeting to achieve sub-logarithmic query time. See Fig. 5. The segment $[a, b]$ is allocated to the nodes colored black and the nodes marked 1, 2. However, only the nodes marked 1, 2 have maximal points. Thus, if we visit the nodes colored black, then we loose our query time.

Clearly different techniques are required to achieve the sub-logarithmic query time bound.

5. The word-RAM data structure for Problem 1

Before we present the data structure, we need to solve two subproblems which form the core components to identify the *profitable nodes* in the main solution.

5.1. Subproblems

Subproblem 1. Given an unsorted array A of n integers from the range of $[0, \sqrt{\log n} - 1]$, preprocess A into a linear space data structure such that given two indices i, j and two values, $a, b : (a, b) \in [0, \sqrt{\log n} - 1] \times [0, \sqrt{\log n} - 1]$, we can efficiently report the smallest value $t \in A[i, j]$ for $a \leq t \leq b$.

This is clearly a variant of range successor problem and we can solve the problem as follows:

Preprocessing:

1. For the element x stored at the ith index of the array A, we create a 2-d point (x, i). Let S' be the set of such points.
2. We preprocess S' into a data structure denoted by *RS*. *RS* is an instance of the data structure of Theorem 4. Remember that, from [17], we know that range successor queries of the form $[x_1, \infty) \times [y_1, y_2] : x_1 \in [1, \frac{\log n}{\log\log n}]$, $(y_1, y_2) \in [1, n] \times [1, n]$ can be answered in $O(1)$ time using a linear space data structure for a set of n 2-d points with coordinates from an integer grid of $[1, \frac{\log n}{\log\log n}] \times [1, n]$. However, in our case, for any point $(x, i) \in S'$, $x \in [0, \sqrt{\log n} - 1]$ and $i \in [1, n]$.

Query algorithm: Our query algorithm is as follows:

1. Given the two indices $i, j : (i, j) \in [1, n] \times [1, n]$ and two values $a, b : (a, b) \in [0, \sqrt{\log n} - 1] \times [0, \sqrt{\log n} - 1]$, we construct a three sided query rectangle of the form $[a, \infty) \times [i, j]$.
2. We run the range successor query on the data structure *RS* and find the smallest value t in $[a, \infty) \times [i, j]$. If $t \leq b$, we return t, else we return *null*.

Lemma 9. *The total storage space needed by RS is $O(|A|)$ words.*

Proof. The number of 2-d points formed is equal to the number of elements in the array A. Thus S' has $O(|A|)$ points. As *RS* is an instance of the data structure of [17], which is a linear space data structure, the total storage space needed by *RS* is $O(|A|)$ words. □

Lemma 10. *The query algorithm reports the smallest element $t : a \leq t \leq b$ for $t \in A[i, \ldots, j]$.*

Proof. As Subproblem 1 is an instance of the range successor query with points restricted in the range of $[1, \sqrt{\log n}] \times [1, n]$, the correctness of our algorithm follows from the correctness of the data structure of [17]. □

Lemma 11. *The query algorithm runs in $O(1)$ time.*

Proof. As Subproblem 1 is an instance of the range successor query with points restricted in the range of $[1, \sqrt{\log n}] \times [1, n]$, and as the data structure of [17] takes $O(1)$ time to answer a range successor query, our query algorithm takes $O(1)$ time. □

Therefore, we conclude:

Theorem 5. *Given an unsorted array A of n integers from the range of $[0, \sqrt{\log n} - 1]$, we can preprocess A into a linear space data structure such that given two indices i, j and two values, $a, b : (a, b) \in [0, \sqrt{\log n} - 1] \times [0, \sqrt{\log n} - 1]$, we can report in $O(1)$ time the smallest value $t \in A[i, j]$ for $a \le t \le b$.*

Subproblem 2. Given an unsorted array A with n integers in the range of $[0, \sqrt{\log n} - 1]$, preprocess A into a data structure such that given an element $x \in [0, \sqrt{\log n} - 1]$ and an index i, we can efficiently report the number of occurrences of $x \in A[0, i - 1]$.

Preprocessing:

1. For each element $x \in [0, \sqrt{\log n} - 1]$, we construct a bit string of size $|A|$ bits such that the bit string supports *rank* query. We denote the string by s_x.
2. The $(n - 1 - l)$th most significant bit of the string s_x is set to "one" if the element stored at $A[l]$ is x.

Lemma 12. *The storage space needed by the above data structure is $O(|A|)$ words.*

Proof. As there are $\log n$ integer elements in the range $[0, \sqrt{\log n} - 1]$, there are $\sqrt{\log n}$ bit strings each of size $|A|$ bits. Thus, the total number of bits needed to store all the strings is $O(|A|\sqrt{\log n}) \le O(|A| \log n)$. Hence the claim holds. □

Query algorithm: Our query algorithm is as follows:

1. Given an element $x \in [0, \sqrt{\log n} - 1]$ and the index i, we perform the rank query $count = rank(s_x, i - 1)$.
2. We return the value of rank.

Lemma 13. *The query algorithm returns the number of occurrences of $x \in A[0, i - 1]$.*

Proof. The string s_x has its $(n - 1 - l)$th most significant bit set to one if $A[l] = x$. Our query returns the count of the number of ones in the binary string s_x by executing *rank* query. Thus, the correctness of our algorithm follows from the correctness of the *rank* query of [5]. □

Lemma 14. *The query algorithm takes $O(1)$ time.*

Proof. Follows directly from Lemma 3. □

We therefore conclude,

Theorem 6. *Given an unsorted array A with n integers in the range of $[0, \sqrt{\log n} - 1]$, A can be preprocessed into a data structure of size $O(n)$ words such that given an element $x \in [0, \sqrt{\log n} - 1]$ and an index i, we can report in $O(1)$ time the number of occurrences of $x \in A[0, i - 1]$.*

5.2. Construction of the main data structure

1. Construct a primary tree T_x, the leaves of which store the x-coordinates of the points in S sorted in increasing order of its values from left to right.
2. Each internal node of the tree T_x has $\sqrt{\log n}$ children which are numbered from 0 to $\sqrt{\log n} - 1$ with the rightmost being 0 and the leftmost being $\sqrt{\log n} - 1$.
3. Each internal node μ has an auxiliary array A_μ which stores the y-coordinates of the points present at the leaves of the subtree rooted at μ. A_μ is sorted in non-increasing order of the y-coordinates. Notice that $A_\mu = \bigcup_{i=0}^{\log n - 1} A_{v(i)}$ for $v(i)$ being a child of $\mu \in T_x$. Therefore, each element $y \in A_{v(i)}$ points to its corresponding position in A_μ.
4. We also maintain an array A'_μ whose ith index stores an integer $j \in [0, \sqrt{\log n} - 1]$ if the value in $A_\mu[i]$ is present in $A_{v(j)}$ for $v(j)$ being a child of μ. Remember that, by assumption, each point in R has distinct x- and y-coordinates. Thus, if the value in $A_\mu[j]$ is present in $A_{v(j)}$, it cannot be present in any other array associated with any other children of the μ. We preprocess the array A'_μ into instances of the data structures of Theorem 5 and Theorem 6.
5. We then create a separate array B_μ whose ith index stores the x-coordinate of the point $p = (p(x), p(y))$ if $p(y)$ is stored in $A_\mu[i]$. We preprocess B_μ to an instance of the data structure of [8] such that given two indices i, j, we can perform $RMQ(B_\mu[i, j])$ in $O(1)$ time.
6. At the node μ, we additionally maintain a tree VT_μ which is an instance of the van Emde Boas tree [16]. VT_μ is constructed on the values of A_μ. A set of n integers can be preprocessed in a van Emde Boas tree such that we can search for a particular integer value in the data set in $O(\log \log n)$ time.

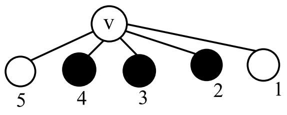

Fig. 6. The segment $[a, b]$ is allocated to the nodes colored black.

Lemma 15. *The height of the tree of is* $O(\frac{\log n}{\log\log n})$.

Proof. Since the degree of each internal node is $\sqrt{\log n}$, the height of the tree of is $O(\frac{\log n}{\log\log n})$. $\square$

Lemma 16. *The storage space needed by the above data structure is* $O(n\frac{\log n}{\log\log n})$ *words.*

Proof. The primary tree T_x needs $O(n)$ words. Each internal node μ maintains three auxiliary arrays A_μ, A'_μ, B_μ. Any element present in A_μ, A'_μ or B_μ cannot be present in any other array A_χ, A'_χ or B_χ for μ, χ being nodes at the same depth h of the tree T_x. Thus, the total size needed by all A_μ, A'_μ or B_μ arrays across all the internals nodes at a particular depth h of T_x is $O(n)$ words. Since the height of T_x is $O(\frac{\log n}{\log\log n})$, the total storage space needed by the above data structure is $O(n\frac{\log n}{\log\log n})$ words. $\square$

5.3. *Emulating fractional cascading for a node with* $\sqrt{\log n}$ *children*

See Fig. 6. Let the children for the node v that are colored black are the ones to which segment $[a, b]$ of the query rectangle $[a, b] \times [c, d]$ is allocated. Let us call the node v as a *group leader* for the set of the nodes $\{2, 3, 4\}$. Also let i, j be the indices for the array A_v such that $A_v[i, j] \in [c, d]$. We now want to visit the rightmost black child $v(i) \in \{2, 3, 4\}$ of v such that $A_{v(i)}$ has elements in $A_v[i, j]$. Remember, by step 4, the array $A'_v[i]$ stores the identity of the array from which the element $A_v[i]$ is coming. As the array A'_v is preprocessed into instances of the data structures of Theorems 5 and 6, we run the range successor query with the query range $[2, \infty) \times [c, d]$ to find the smallest element $t \in A'_v[i, j]$ such that $2 \le t$. If $t \le 4$, we run the following two *rank* queries with the binary string s_t and find $i' = rank(s_t, i - 1)$ and $j' = rank(s_t, j)$. We thus get two indices $i', j' : A_t[i', j'] \in [d, c]$. However, if $t > 4$, then it means that there are no elements in $A_v[i, j]$ that are coming from A_2, A_3 or A_4.

Lemma 17. *For the node v of Fig. 6, we can find the rightmost canonical node with points in the query rectangle in $O(1)$ time.*

Proof. Follows directly from Theorem 5 and Theorem 6. $\square$

5.4. *Query algorithm*

1. Given a query rectangle $[a, b] \times [c, d]$, the segment $[a, b]$ is allocated to a node $\mu \in T_x$ if $int(\mu) \subset [a, b]$ but $int(parent(\mu)) \not\subseteq [a, b]$. There will be $O(\frac{\log^{3/2} n}{\log\log n})$ (see [13]) such nodes. Denote the set of such nodes as V.
2. As stated in [13], the set of all such canonical nodes can be grouped into $O(\frac{\log n}{\log\log n})$ groups with each group g_i containing some children $v_l \dots v_k$ for some node v. We will denote the node v as *group leader* $GL(i)$. Let $G = \{GL(1), \dots, GL(O(\frac{\log n}{\log\log n}))\}$ be the set of *group leaders* stored in order of their positions from right to left in the tree T_x. The exact details of finding the group leaders and the canonical nodes follow next.
3. Find the least common ancestor for the values a, b in T_x.
4. Search for the smallest value $\ge c$ and the largest value $\le d$ in the auxiliary array A associated with the least common ancestor. The smallest and the largest values can be found in $O(\log\log n)$ time by searching the corresponding van Emde Boas tree associated with the least common ancestor.
5. Next, by using the technique for Subproblem 2 and the technique discussed in the previous subsection 5.3, we can find the appropriate positions for c, d in the auxiliary arrays associated with all the *group leaders* in $O(\frac{\log n}{\log\log n})$ time in total.
6. We start traversing the *group leaders* in order of their positions from right to left in the tree T_x starting from the rightmost node that in the node $GL(1)$.
7. Let the positions for d, c in $A_{GL(1)}$ be i, j. Also, let the nodes $v(p), \dots, v(\sqrt{\log n} - 1)$ be the children of $GL(1)$ to which the segment $[a, b]$ is allocated. It should be remembered that the children of a node are numbered in order of their positions from right to left with the rightmost being numbered 0 and the leftmost being numbered $\sqrt{\log n} - 1$.
8. Perform range successor query with the range $[p, \infty) \times [i, j]$ to find the smallest $t \in [p, \sqrt{\log n} - 1]$ such that $t \in A'_{GL(1)}[i, j]$.

9. Find the indices m, m' of the largest value smaller than or equal to $A_{G(1)}[i]$ and the smallest value greater than or equal to $A_{G(1)}[j]$ in the array A_t, the auxiliary array associated with the node t. This can be done by using the technique of Theorem 6.
10. Run $l = RMQ(B_t[m, m'])$ and report the point whose y-coordinate is stored in $A_t[l]$ as a maximal point. Next, repeat $l' = RMQ(B_t[m, l-1])$ and report the point whose y-coordinate is stored in $A_t[l']$. Repeat similar steps until the point stored at $A_t[m]$ is reported.
11. Let the y-coordinate for the last maximal point reported from the node t be y_1. Find the position j' of y_1 in the array $A_{GL(1)}$. This can be done in $O(1)$ time as there is a pointer from every element to its corresponding position in the auxiliary array associated with the parent node.
12. Repeat range successor query at the node $GL(1)$ with the range $[t+1, \infty) \times [i, j'-1]$. If no other child of $GL(1)$ has any maximal point for q, we move to the next *group leader* $GL(2)$.
13. Repeat similar steps until all the group leaders are visited.

5.4.1. How to find the canonical nodes

1. Given a query rectangle $[a, b] \times [c, d]$, find the leaves in the tree T_x storing the values a, b. As all the points have distinct x- and y-coordinates (by assumption) and as all the points have coordinates in $[1, n] \times [1, n]$, the values a, b are present in the leaves of tree T_x.
2. Start visiting the ancestors for the leaves storing a, b and find the least common ancestor μ for the two leaves. Let π_1 be the path from the leaf storing a to μ.
3. For any node $v_i \in \pi_1 : v_i \neq \mu$, check if $int(v_i) \subset [a, b]$. If "true", then:
   (a) check if $int(parent(v_i)) \subseteq [a, b]$. If "yes", then move to $parent(v_i)$.
   (b) Else, the nodes $v_i, \ldots, v_0$ are the canonical nodes and the node $parent(v_i)$ is the group leader node.
4. If $int(v_i) \nsubseteq [a, b]$, then the nodes $v_{i-1}, \ldots, v_0$ are the canonical nodes and the node $parent(v_i)$ is the group leader node.
5. If $parent(v_i)$ is a group leader, then denote the node as $GL(i)$. Let $v_i, v_{i-1}, \ldots, v_0$ be the canonical nodes. Also let $int(v_i) = [x_1, x_2]$ and $int(v_0) = [x_3, x_4]$. Along with the node $GL(i)$ maintain a tuple denoted by $Tuple_{GL(i)}$ in which we store $\langle v_i, v_0, [x_1, x_4] \rangle$.
6. We perform similar steps for the nodes on the path π_2 from the leaf storing b to the least common ancestor μ. For any node $v_j \in \pi_2$, if $int(parent(v_j))$ is not contained in $[a, b]$, then either $\{v_{\sqrt{\log n}-1}, \ldots, v_j\}$ are the canonical nodes or $\{v_{\sqrt{\log n}-1}, \ldots, v_{j+1}\}$ are the canonical nodes.
7. If $v_i = \mu$, then find the children v_u, v_w of v_i such that $v_u \in \pi_1$ and $v_w \in \pi_2$. If $int(v_u) \in [a, b]$ and/or $int(v_w) \in [a, b]$, then v_i is the group leader node and v_u and v_w are the leftmost and rightmost children of v_i that are canonical nodes. Else if $|u - w| > 1$, there is at least one child of v_i which is a canonical node. We maintain a similar tuple as discussed above if in case the least common ancestor is a group leader.
8. Notice that these group leader nodes can be sorted easily in order of their positions from right to left in the tree T_x. As we are moving form bottom to top for the path π_2, the first group leader node on the path π_2 is the rightmost group leader node. The next group leader node on the path π_2 is the second rightmost group leader node and so on.

Lemma 18. *Any node visited by our query algorithm is either a canonical node or a group leader.*

Proof. Any node visited either on path π_1 or π_2 is a group leader node. Any other node which is not present either in π_1 or in π_2 is visited only if it has been reported by a range successor query at a group leader. For any group leader node, we have a tuple containing the identity of its rightmost and its leftmost children that are canonical nodes. For any node reported by a range successor query, it is ensured that it is in between the leftmost and the rightmost children stored in the tuple. Hence, any node visited by our query algorithm is either a canonical node or a group leader. □

Lemma 19. *For any canonical node ϕ visited by the query algorithm, we have the correct indices i, j for the array A_ϕ such that $A_\phi[i, j] \in [d, c]$.*

Proof. Let us assume that we have the correct indices i, j for the parent of ϕ which is denoted by $parent(\phi)$. We also assume that the range successor query at $parent(\phi)$ returned ϕ to be the next profitable node to visit. Then, the value ϕ must be present in $A'_{parent(\phi)}[i, j]$. Thus, the bit string s_ϕ must have at least one bit set in between $(n-i)$th most significant bit (msb) and $(n-j)$th most significant bit. If we execute $l = rank_1(s_\phi, n-i+1)$, then we get the number of 1s in s_ϕ till the $(i-1)$th msb. Thus, the element $A_\phi[l+1]$ is the first value in A_ϕ to be present in the range of $[d, c]$. Thus, the correctness of our technique is dependent on the correctness of the rank queries and the correct indices for the auxiliary array $A_{parent(\phi)}$ associated with the parent of ϕ. The correctness of the rank queries follow from [5]. So we have to prove the correctness of the indices for the auxiliary array $A_{parent(\phi)}$ associated with the parent of ϕ. Notice that we first search the auxiliary array A associated with the least common ancestor to find the indices i, j containing the largest value smaller than or equal to d and the smallest value greater than or equal to c respectively. Notice that any group leader node is either on the path π_1 from the leaf storing a to the lca or on the path π_2 from the leaf storing b to the lca. If nodes on the path π_1 (or π_2) are arranged in decreasing order of their depths at which these nodes are present (assuming leaves to be at depth zero), then the ith group leader in this sorted arrangement is the parent of the $(i+1)$th group leader. Thus, by performing two rank queries at the ith group leader, we can find the indices for $(i+1)$th group leader and the correctness of these indices at the $(i+1)$th node is ensured by the correctness of the rank queries [5]. Also, each element in A_{i+1} has a pointer to its corresponding position in A_i. Thus, for the element $A_{i+1}[j]$, we can find in $O(1)$ time the largest element smaller than $A_{i+1}[j]$ in A_i. Hence the claim holds. □

Lemma 20. *The query algorithm reports the points in nondecreasing order of y-coordinates.*

Proof. By preprocessing step 1, the array A associated with any internal node is sorted in nonincreasing order. For any canonical node $\phi : A_\phi[i, j] \in [d, c]$ if ϕ is the rightmost canonical node among all the canonical nodes, then we first find the index l of point with maximum x-coordinate such that $i \le l \le j$. Subsequently, in the next iteration, we repeat range maxima to find the point with maximum x-coordinate among all the points with y-coordinates in $A_\phi[i, l-1]$ and so on. As $A_\phi[i] > A_\phi[i+1], \dots, A_\phi[j]$, the points reported from ϕ are in nondecreasing order of y-coordinates. However, if ϕ, the next canonical node to be visited is not the rightmost canonical node, then we first find the smallest value $> p_y$ where p_y is the y-coordinate of the last reported maximal point and then we first report the point with maximum x-coordinate above p_y but less than or equal to d. We continue to repeat similar steps as discussed here. Hence the claim holds. □

Lemma 21. *Any point reported by the query algorithm is a maximal point.*

Proof. Follows directly from Lemma 20. □

Lemma 22. *Any canonical node visited by our query algorithm has at least one maximal point in $[a, b] \times [c, d]$.*

Proof. Before visiting any canonical node ϕ, we visit its parent which is a group leader node $GL(m)$ and execute a range successor query to decide the next rightmost canonical node that has elements in $A_{GL(m)}[i, j-1]$ where $A_{GL(m)}[i]$ is the largest element smaller than the value d and $A_{GL(m)}[j-1]$ is the smallest element greater than the y-coordinate of the last reported maximal point or the value c. The range successor query returns ϕ only if ϕ is present in $A'_{GL(m)}[i, j-1]$ and ϕ is a canonical node. Notice that by step 4 of construction of the main data structure, the array $A'_{GL(m)}$ is created in accordance with the array $A_{GL(m)}$.

As $A_{GL(m)}$ is sorted in non-increasing order of y-coordinates, any element $y \in A_{GL(m)}[i, j-1]$ cannot be dominated by any previously reported maximal points. Thus, if $A_{GL(m)}[i, j-1]$ has elements from A_ϕ, then the node ϕ has maximal points for $[a, b] \times [c, d]$. □

Lemma 23. *All the maximal points in the query range are reported by the query algorithm.*

Proof. If a point p is a maximal point, then p must be present in the subtree rooted at some canonical node ϕ. The corresponding group leader node $GL(m)$ which is parent of ϕ must be present in S_{can}. By step 13 the query algorithm visits all the nodes present in S_{can}. By step 8 of the query algorithm, for any node in S_{can} we execute range successor query to find the current rightmost canonical node. By step 12, once all the children of the current group leader that have maximal points for the query rectangle are visited, we move to the next node next group leader in the S_{can}. At each canonical node we report the maximal points by repeating range maxima queries. Thus, reporting the point p is ensured by the correctness of range maxima data structure of [8] while visit to the node ϕ is ensured by the correctness of the range successor query of [17] and visiting the group leader node $GL(m)$ is ensured by the step 13 of the query algorithm. □

Lemma 24. *The query algorithm reports all the maximal points inside the query rectangle in time $O(\frac{\log n}{\log\log n} + k)$ where k is the size of the output.*

Proof. The total run time of our query algorithm is dependent on three factors namely: (i) the number of *group leaders* we visit; (ii) the number of canonical nodes we visit and (iii) the number of maximal points we report. From [13], it is known that the number of *group leaders* is $O(\frac{\log n}{\log\log n})$. A particular child of a *group leader* is visited if the segment $[a, b]$ is allocated to the node and the node holds at least one maximal point for the query rectangle. While at a *group leader*, we decide in $O(1)$ time, the next child node of the *group leader* that we should visit to report maximal points (check step 8 of the query algorithm). Thus, the number of children of the *group leaders* we visit is $\le k$, the number of maximal points in the query rectangle. Next, we need to prove that (i) we can visit all the *group leaders* in $O(\frac{\log n}{\log\log n})$ time and (ii) the time we spend at a child of a *group leader* is equal to the number of maximal points reported from that node. We start with the first part. It should be noted that for any *group leader* node, one of the following two conditions has to be true: (a) it is present in the path from the least common ancestor to the node $GL(1)$; (b) it is present in the path from the least common ancestor to the node $GL(O(\frac{\log n}{\log\log n}))$.

Moving from a particular node to its parent needs $O(1)$ time following the step 11 of the query algorithm. Similarly moving from a parent to any of its child needs $O(1)$ time following the step 5 of the query algorithm. Thus, the total time needed to visit all the *group leader* nodes is equal to the maximum height of either of the two paths which is $O(\frac{\log n}{\log\log n})$. At each *group leader*, we decide in $O(1)$ time the next child of its that we should visit. At any child of the *group leader* that we visit, we repeat range maximum queries until there are no maximal points from that node. Each run of range maxima query needs $O(1)$ time. Thus, the total time spent in the child is equal to the number of maximal points reported from that node. Therefore, the query algorithm reports all the maximal points inside the query rectangle in time $O(\frac{\log n}{\log\log n}+k)$ where k is the size of the output. $\square$

Combining Lemma 16 and Lemma 24, we conclude to Theorem 1.

6. Solution for Problem 2

6.1. Outline of the solution

As stated earlier, the number of canonical nodes with maximal points in a query rectangle can be $O(\frac{\log^{3/2} n}{\log\log n})$. However, we are targeting a sub-logarithmic query time algorithm. Hence, we will devise a technique to count the number of maximal points coming from the children of a group leader for a query rectangle without visiting its children explicitly. But before doing so, we will first study the problem in a restricted domain which is defined next.

6.2. A subproblem

In order to solve Problem 2, we will need an efficient solution for the following subproblem.

Subproblem 3. Given a set S of n points from a universe of $[1,\log^{\rho} n]\times[1,n]$ for $0<\rho\le\frac{1}{2}$, preprocess the points into a data structure such that given an axis parallel rectangle $q=[a,b]\times[c,d]$ for $(a,b)\in[1,\log^{\rho} n]\times[1,\log^{\rho} n]$ and $(c,d)\in[1,n]\times[1,n]$, we can efficiently count the number of maximal points in $S\cap q$.

In the complete solution, an efficient technique for the above problem will help us to count in $O(1)$ time the number of maximal points in q that are coming from children of a group leader.

Solution for Subproblem 3

Preprocessing:

1. First of all, store the y-coordinates of the points in an array A. Then, construct a height-balanced binary search tree T_y whose leaf nodes store the values in the array A and are at the same level. At each internal node $m\in T_y$, store a key value y_m which is the median of the points stored in the subtree rooted at m.
2. For each pair of possible points $(i_1,i_2)\in[1,\log^{\rho} n]\times[1,\log^{\rho} n]$, form an interval $[i_1,i_2]$ (including $i_1=i_2$).
3. For each value $y_2\in A$ and for each possible interval $[i_1,i_2]$, form 3-sided anchored rectangles $[i_1,i_2]\times[y_2,\infty)$ and $[i_1,i_2]\times(-\infty,y_2]$. Next, for the interval $[i_1,i_2]$ and the value y_2, do the following:
   (a) Visit the ancestors of y_2 in the tree T_y. At each ancestor m, find the key value y_m.
   (b) If $y_m<y_2$, form an axis-parallel rectangle $R_1=[i_1,i_2]\times[y_m,y_2]$.
      i. For the points in $S\cap R_1$, compute the subset of points that are not dominated by any other point. We call such a subset a maximal chain. Let p_{ymax} and p_{xmax} respectively be the topmost point and the bottommost point of the maximal chain.
      ii. Count the number of maximal points in the chain p_{xmax} to p_{ymax}. Denote the value as $|p_{xmax},p_{ymax}|$. Store the value in a variable denoted by $count_m(y_2)$.
      iii. Next, for the point p_{xmax}, find the topmost point p_{nodom} in $[i_1,i_2]\times(-\infty,y_m]$ such that p_{nodom} is not dominated by p_{xmax}.
      iv. Create a tuple $\langle count_m(y_2),p_{nodom}\rangle$ and store it with reference to the rectangle R_1 in a lookup table. Here the suffix m denotes the index for the node $m\in T_y$ that is an ancestor of the leaf node storing the value y_2.
      v. **Special cases**
         A. If no points are present in the rectangle R_1, store $\langle 0,NULL\rangle$.
         B. If the point p_{nodom} does not exist, store $\langle count_m(y_2),NULL\rangle$.
   (c) On the other hand, if $y_m>y_2$, form a rectangle $R_2=[i_1,i_2]\times[y_2,y_m]$ and then find the topmost point p'_{ymax} and the bottommost point p'_{xmax} in the maximal chain for the points in $S\cap R_2$. Count the number of maximal points in the chain from p'_{xmax} to p'_{ymax} that is count $|p'_{xmax},p'_{ymax}|$. Store it in a tuple $\langle count'_m(y_1),p'_{xmax}\rangle$.

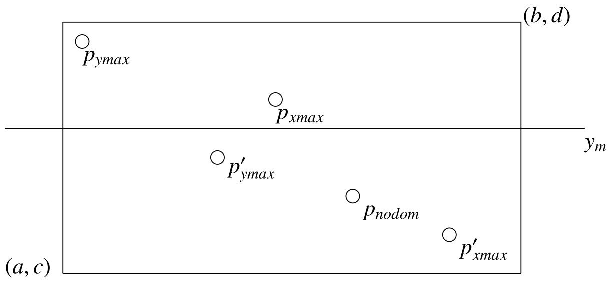

Fig. 7. The rectangle $[a,b]\times[c,d]$ is split into two parts (a) $[a,b]\times[y_m,d]$ and (b) $[a,b]\times[c,y_m]$. The points p_{xmax}, p_{ymax} (respectively p'_{xmax}, p'_{ymax}) are the bottommost and the topmost points of the maximal chain in $[a,b]\times[y_m,d]$ (respectively $[a,b]\times[c,y_m]$). The point p_{nodom} is the topmost point in the anchored rectangle $[a,b]\times(-\infty,y_m)$ which is not dominated by p_{xmax}.

(d) **Special case**
  i. If there are no points in R_2, store $\langle 0, NULL\rangle$.

4. Maintain the data structure of [9] to find the least common ancestor for two given leaf nodes of the tree T_y in $O(1)$ time.

Lemma 25. *The total storage space needed by the data structure is $O(n\log^{1+2\rho} n)$ words.*

Proof. As $(i_1,i_2)\in[1,\log^\rho n]\times[1,\log^\rho n]: 0<\rho\le\frac{1}{2}$, the number of possible intervals formed in step 2 is $O(\log^{2\rho} n)$. With n possible values of $y\in[1,n]$, the number of three sided rectangles formed is $O(n\log^{2\rho} n)$. For each three sided rectangle, we form $O(\log n)$ four sided rectangles in step 3. With each four sided rectangles, a tuple is stored. Thus, there are $O(n\log^{1+2\rho} n)$ tuples. Any tuple is of at most $3\log n$ bits which is equal to $O(1)$ words. Thus, the total storage space is $O(n\log^{1+2\rho} n)$ words. □

Query algorithm

1. Given a query rectangle $[a,b]\times[c,d]$ such that $(a,b)\in[1,\log^\rho n]\times[1,\log^\rho n]$ for $0<\rho\le\frac{1}{2}$ and $(c,d)\in[1,n]\times[1,n]$, find the least common ancestor m for the values d, c in the tree T_y. Let the key value stored at m be y_m.
2. Consider the rectangle $[a,b]\times[y_m,d]$ and the corresponding tuple $\langle count_m(d), p_{nodom}\rangle$. The suffix m has the same meaning as specified in step 3(b)iv of the preprocessing algorithm.
3. If $p_{nodom}\neq NULL$:
   Let $p_{nodom}(y)$ be the y-coordinate of the point p_{nodom}.
   (a) If $c\le p_{nodom}(y)$, we do the following:
      i. Consider the rectangle $[a,b]\times[p_{nodom}(y), y_m]$. Choose the value $count'_m(p_{nodom}(y))$ from the corresponding tuple for the rectangle $[a,b]\times[p_{nodom}(y), y_m]$.
      ii. For the rectangle $[a,b]\times[c,y_m]$ choose the value $count'_m(c)$.
      iii. Return $count'_m(c) - count'_m(p_{nodom}(y)) + 1 + count_m(d)$.
   (b) Else, (that is $p_{nodom}(y) < c$):
      i. Return the value $count_m(d)$.
4. Else, (that is if $p_{nodom} = NULL$):
   (a) If $count_m(d)\neq 0$, return $count_m(d)$.
   (b) Else, return $count'_m(c)$.

6.3. Analysis of the query algorithm

Lemma 26. *Let $p_{nodom}\neq NULL$ and $c\le p_{nodom}(y)<d$. Then, the point p_{nodom} belongs to the subtree rooted at $m\in T_y$.*

Proof. Since $c\le p_{nodom}(y)\le d$ and m is the least common ancestor of c and d in T_y, the point p_{nodom} belongs to the subtree rooted at $m\in T_y$. □

Lemma 27. *Let $p_{nodom}\neq NULL$ and $c\le p_{nodom}(y)<d$. Then, the maximal chain in the rectangle $[a,b]\times[c,y_m]$ will pass through the point p_{nodom}.*

Proof. As the point p_{nodom} is the topmost point below y_m and above c that is not dominated by p_{xmax}, the point with maximum x-coordinate in $[a,b]\times[y_m,d]$, the point p_{nodom} cannot be dominated by any other point in the rectangle in $[a,b]\times[c,y_m]$ as that point dominating p_{nodom} in $[a,b]\times[c,y_m]$ would have been the topmost point not dominated by p_{xmax}. □

Lemma 28. *See Fig. 7. Let p'_{xmax} and p'_{ymax} be respectively the two points with maximum x- and y-coordinates in the rectangle $[a,b] \times [c, y_m]$. Let $|p'_{xmax}, p_{nodom}|$ denote the number of maximal points between p'_{xmax} and p_{nodom} (including p'_{xmax} and p_{nodom}) inside the rectangle $[a,b] \times [c, y_m]$. Then $|p'_{xmax}, p_{nodom}| = |p'_{xmax}, p'_{ymax}| - |p'_{ymax}, p_{nodom}| + 1$.*

Proof. By Lemma 27, the point p_{nodom} is a point in the maximal chain from p'_{xmax} to p'_{ymax}. As the rectangle $[a,b] \times [p_{nodom}(y), y_m]$ is contained in $[a,b] \times [c, y_m]$, any maximal point in $[a,b] \times [p_{nodom}(y), y_m]$ is also a maximal point in $[a,b] \times [c, y_m]$. Therefore, $|p'_{xmax}, p_{nodom}| = |p'_{xmax}, p'_{ymax}| - |p'_{ymax}, p_{nodom}| + 1$. □

Lemma 29. *If $c \le p_{nodom}(y) < d$, then the rectangle $[a,b] \times [p_{nodom}(y), y_m]$ is present in the set of $O(n \log^{1+2\rho} n)$ rectangles for which we precomputed the count of maximal points.*

Proof. As $c \le p_{nodom}(y) < y_m \le d$, the leaf storing the value $p_{nodom}(y)$ is present in the subtree rooted at the internal node ϕ storing the value y_m. Hence the node ϕ is an ancestor for the leaf storing $p_{nodom}(y)$. Hence the claim holds. □

Lemma 30. *Let $p_{nodom} \neq NULL$ and $c \le p_{nodom}(y) < d$. Then, the number of maximal points inside the rectangle $[a,b] \times [c,d]$ is equal to the number of maximal points in the rectangle $[a,b] \times [c, p_{nodom}(y)]$ plus the number of maximal points in $[a,b] \times [p_{xmax}(y), d]$.*

Proof. As the point p_{nodom} is not dominated by p_{xmax}, $p_{nodom}(x) > p_{xmax}(x)$. Here $p_{nodom}(x)$ and $p_{xmax}(x)$ are the x-coordinates of the points p_{nodom} and p_{xmax} respectively. Any point in the maximal chain inside the rectangle $[a,b] \times [c, p_{nodom}(y)]$ must have an x-coordinate greater than $p_{nodom}(x)$, otherwise the point will be dominated by p_{nodom}. Also, it should be noted that any point in the maximal chain in the rectangle $[a,b] \times [p_{xmax}(y), d]$ is not dominated by any other point in the rectangle $[a,b] \times [p_{xmax}(y), d]$ or $[a,b] \times [c, p_{xmax}(y)]$. Hence, the number of maximal points inside the rectangle $[a,b] \times [c,d]$ is equal to the number of maximal points in the rectangle $[a,b] \times [c, p_{nodom}(y)]$ plus the number of maximal points in $[a,b] \times [p_{xmax}(y), d]$. □

Lemma 31. *Let $p_{nodom} \neq NULL$ but $p_{nodom}(y) < c$. The number of maximal points inside the rectangle $[a,b] \times [c,d]$ is equal to the number of maximal points in the rectangle $[a,b] \times [y_m, d]$.*

Proof. As p_{nodom} is the topmost point not dominated by p_{xmax} below y_m, the case of $p_{nodom}(y) < c$ is possible only if (a) there are no points in the rectangle $[a,b] \times [c, y_m]$ or (b) any point in the rectangle $[a,b] \times [c, y_m]$ is dominated by the point p_{xmax}. In any case, no maximal point for the rectangle $[a,b] \times [c,d]$ is present in $[a,b] \times [c, y_m]$. Hence the claim holds. □

Lemma 32. *If $p_{nodom} = NULL$ but $count_m(d) \neq 0$, the number of maximal points in $[a,b] \times [c,d]$ is equal to $count_m(d)$.*

Proof. If $p_{nodom} = NULL$, then there is no point in $[a,b] \times (-\infty, y_m]$ not dominated by p_{xmax}, the point with maximum x-coordinate in $[a,b] \times [y_m, d]$. Hence, the number of maximal points in $[a,b] \times [c,d]$ is equal to $count_m(d)$. □

Lemma 33. *If $p_{nodom} = NULL$ and $count_m(d) = 0$, the number of maximal points in $[a,b] \times [c,d]$ is equal to $count'_m(c)$.*

Proof. By step 3(b)vA of preprocessing, if $p_{nodom} = NULL$ and $count_m(d) = 0$, then the rectangle $[a,b] \times [y_m, d]$ is empty. Hence, the number of maximal points in $[a,b] \times [c,d]$ is equal to $count'_m(c)$. □

Lemma 34. *For any query rectangle $[a,b] \times [c,d] : (a,b) \in [1, \log^\rho n] \times [1, \log^\rho n]$, $(c,d) \in [1,n] \times [1,n]$ and $0 < \rho \le \frac{1}{2}$, our query algorithm correctly counts the number of maximal points in the query rectangle.*

Proof. Given the query rectangle $[a,b] \times [c,d]$, we first find the value y_m such that $c \le y_m \le d$ and y_m is stored in the least common ancestor for the leaves storing c, d. We split the rectangle $[a,b] \times [c,d]$ into two smaller rectangles $R_1 = [a,b] \times [c, y_m]$ and $R_2 = [a,b] \times [y_m, d]$. For the rectangle R_2, we consider its corresponding tuple and find the topmost point p_{nodom} in $[a,b] \times (-\infty, y_m]$ not dominated by any maximal point in $[a,b] \times [y_m, d]$. If $p_{nodom} \neq NULL$ and $p_{nodom} \ge c$, then Lemmas 28, 29 and 30 ensure that the correct result is returned. However, if $p_{nodom} = NULL$, then Lemmas 31 and 33 ensure that the correct result is returned. □

Query time analysis

Lemma 35. *The query algorithm takes $O(1)$ time to count the number of maximal points inside a given query rectangle.*

Proof. As all n points have distinct y-coordinates and the y-coordinates of the points are in the range $[1,n]$, the values c, d will be present in array A (see step 1 of preprocessing). Thus, locating the indices (as well as the leaf nodes) storing these two values can be done in $O(1)$ time. Finding the least common ancestor m can also be done in $O(1)$ time using the data structure of [9]. Thus, all we are left with is to find the tuples for the rectangles $[i_1, i_2] \times [c, y_m]$ and $[i_1, i_2] \times [y_m, d]$ by using the lookup table and then adding the respective counts. All these operations can be done in $O(1)$ time. □

By Lemma 25 and Lemma 35, we conclude:

Theorem 7. *Given a set S of n points from a universe of $[1, \log^\rho n] \times [1, n]$ for $0 < \rho \le \frac{1}{2}$, we can preprocess the points into a data structure of size $O(n \log^{1+2\rho} n)$ words, such that given an axis parallel rectangle $q = [a, b] \times [c, d]$ for $(a, b) \in [1, \log^\rho n] \times [1, \log^\rho n]$ and $(c, d) \in [1, n] \times [1, n]$, we can count the number of maximal points in $S \cap q$ in $O(1)$ time.*

Special note. In the above solution, we have shown that we can preprocess the count of maximal points and do some bookkeeping for some $O(n \log^{1+2\rho} n) : 0 < \rho \le \frac{1}{2}$ rectangles when the horizontal intervals for the query rectangles are bounded in the range of $[1, \log^\rho n]$ and the vertical intervals for the query rectangles are bounded in the range of $[1, n]$. Given a query rectangle, we split the rectangle into two smaller rectangles for which we already have the count of maximal points and some additional information using which we can compute the count of the maximal points in a query rectangle in $O(1)$ time. We will use this idea in the next section.

7. Solution for the general problem

7.1. Preprocessing for the main data structure

7.1.1. Preprocessing phase 1

1. Construct a tree T_x whose leaf nodes are at the same level (height) and store the x-coordinates of the points in the set R in non-decreasing order of their values.
2. Each internal node $\mu \in T_x$ has $O(\sqrt{\log n})$ children, the left most child being numbered as $\sqrt{\log n} - 1$ and the right most child being 0. Each internal node $\mu \in T_x$ is assigned an interval $int(\mu)$ which is equal to the union of the discrete intervals induced on the x-axis by the values stored at the leaf nodes of the subtree rooted at μ.
3. Next, the following arrangement has to be done for all the internal nodes of the tree T_x except the root.
   (a) Each internal node μ has an auxiliary array A_μ which stores the y-coordinates of the points whose x-coordinates are present in the leaf nodes of the subtree rooted at μ. Thus, $A_\mu = \bigcup_{i=0\ldots\sqrt{\log n}-1} A_i$ where i is a child of μ. A_μ is sorted in non-increasing order of its values.
   (b) Each element of $A_i, i = 0, \ldots, \sqrt{\log n} - 1$ will point to its corresponding position in the array A_μ.
   (c) However, there will be no pointers from the elements of the array A_μ to the elements in the arrays of its children. Rather, the array A_μ will be preprocessed into $D_{\mu(1)}$ which is an instance of the data structure of Theorem 5. While constructing the data structure $D_{\mu(1)}$, we have to perform the following step:
      i. For the value y_j stored in $A_\mu[j]$, create a corresponding point (i, y_j) if the value y_j came from the array A_i where i is a child of μ. Thus we have a set of points from a grid of $[0, \sqrt{\log n} - 1] \times [1, n]$.
   (d) Each child v_i of μ maintains a binary string *lookup* of size $|A_\mu|$. The ith most significant bit of the string *lookup* is set to one if $A_\mu[i] \in A_{v_j}$. Notice that $A_\mu = \bigcup A_{v_j}$, for v_j being a child of μ. The string *lookup* should support *rank*() and *select*() (see [5]) queries.
   (e) Maintain a range maximum data structure (see [18]) RM_{A_μ} such that given two indices i, j of A_μ, we can return the maximum x-coordinate among the points whose y-coordinates are stored between $A_\mu[i]$ to $A_\mu[j]$.
   (f) For the values of the array A_μ, construct the following two auxiliary trees at the node μ:
      i. VT_μ which is an instance of the van Emde Boas tree [16]. A set of n integers can be preprocessed in a van Emde Boas tree such that we can search for a particular integer value in the data set in $O(\log \log n)$ time.
      ii. A height balanced binary search tree $T_{\mu,y}$. A node $\phi \in T_{\mu,y}$ stores the median of the values stored in the leaf nodes of its subtree. We denote the value as *key*.
4. For the root we do the following
   (a) Each index i of the array at the root node A_{root} will have $2\sqrt{\log n}$ pointers of which $\sqrt{\log n}$ pointers will be pointing to the smallest elements greater than $A_{root}[i]$ in each of the arrays A_j for j being a child of the root node.
   (b) Similarly the other $\sqrt{\log n}$ pointers will be pointing to the largest elements greater than $A_{root}[i]$ in the arrays A_j for j being a child of the root node.
   (c) Construct a range maximum data structure $RM_{A_{root}}$ such that given two indices i, j of A_{root}, we can return the maximum x-coordinate among the points whose y-coordinates are stored between $A_{root}[i]$ to $A_{root}[j]$.

7.1.2. Preprocessing phase 2

1. At each internal node $\mu \in T_x$ of the data structure we constructed above, we do the following:
   for $i = 0, \ldots, \sqrt{\log n} - 1$ and for $j = 0, \ldots, \sqrt{\log n} - 1$:
   (a) Form a pair (i, j). Consider all the points present in the subtrees rooted at the $v_i, v_{i-1}, \ldots, v_{j+1}, v_j$. Store these points in a set S'.

(b) Form a vertical slab V whose x-interval is equal to $int(v_j) \cup int(v_{j+1}) \cup \ldots \cup int(v_{i-1}) \cup int(v_i)$. We denote this interval as $[x_1, x_2]$.
(c) For each $y \in A_\mu$, form two three sided rectangles of the form $[x_1, x_2] \times (-\infty, y]$ and $[x_1, x_2] \times [y, \infty)$.
(d) Visit the ancestors for the leaf storing the value y in the tree $T_{\mu,y}$. If the key y_m stored at an ancestor μ is $< y$, form a rectangle $R_1 = [x_1, x_2] \times [y_m, y]$.
(e) We then find
  i. The topmost and the rightmost points of the maximal chain in $R_1 \cap S'$. Denote these points as (p_{top}, p_{bottom}).
  ii. The count of the maximal points in $R_1 \cap S'$. Also find the topmost point in $[x_1, x_2] \times (-\infty, y_m]$ which is not dominated by p_{bottom}. Denote the point as p_{nodom}. If there are no such points, we store p_{nodom} as *NULL*.
(f) This information is stored in a tuple $Tuple_{R_1} = \langle p_{top}, p_{bottom}, count, p_{nodom} \rangle$ for the rectangle R_1.
(g) However, if the key $y_m > y$, we then form the rectangle $R_2 = [x_1, x_2] \times [y, y_m]$ and find the count of maximal points in $R_2 = [x_1, x_2] \times [y, y_m]$ and store it in a tuple $Tuple_{R_2}$. Also store in the tuple, the topmost and the rightmost points in R_2.

Given a query rectangle $[a, b] \times [c, d]$, for any group leader $GL(i)$ if $v_i, v_{i-1}, \ldots, v_j : i > i-1 > \ldots, j$ are its children that are canonical nodes, then $int(v_i) \subset [a, b], int(v_{i-1}) \subset [a, b], \ldots, int(v_j) \subset [a, b]$. Let the horizontal interval $[x_1, x_2]$ be equal to $\bigcup_{m=j}^{i} int(v_m)$. Thus, if we have (i) the count of the maximal points for the rectangles $[x_1, x_2] \times [c, y_m]$ and $[x_1, x_2] \times [y_m, d] : c \le y_m \le d$ and (ii) additional information like the topmost point, rightmost point in $[x_1, x_2] \times [y_m, d]$ and the topmost in $[x_1, x_2] \times (-\infty, y_m]$, we can then compute the count of the maximal points coming from the children of $GL(i)$ for the rectangle $[a, b] \times [c, d]$ in $O(1)$ time. That is the reason for the construction of the auxiliary data structure at the node μ at preprocessing phase 2. Notice that this data structure is an instance of the data structure of Theorem 7 used in the previous section for counting maximal points in a restricted setting.

Lemma 36. *The total storage space needed by the data structure for the counting problem will be* $O(n\frac{\log^3 n}{\log\log n})$.

Proof. The height of the primary tree T_x is $O(\frac{\log n}{\log\log n})$. Each internal node $\mu \in T_x$ of the tree has a degree of $O(\sqrt{\log n})$. Also, the node μ is associated with an auxiliary data structure which is an instance of the data structure of Theorem 7. This auxiliary data structure needs a storage space of $O(|T_{x,\mu}| \log^2 n)$ words where $|T_{x,\mu}|$ is the number of leaves present in the subtree $T_{x,\mu}$ rooted at the node $\mu \in T_x$. Thus, the total storage space needed at a particular level of the tree is $O(n \log^2 n)$. With height of the tree T_x being $O(\frac{\log n}{\log\log n})$, the total storage space needed by the data structure is $O(n\frac{\log^3 n}{\log\log n})$. □

7.2. Query algorithm for the main data structure

1. Given a query rectangle $[a, b] \times [c, d]$, find the group leaders in tree T_x using the steps discussed earlier. Sort these group leaders in order of their positions from right to left in the tree T_x as we visit these group leaders in this order.
2. Let $GL(1)$ be the rightmost group leader node. Visit $GL(1)$ and consider the tuple $Tuple_{GL(1)} = \langle v_{\sqrt{\log n}-1}, v_j, [x_1, x_4] \rangle$ (see step 5 in Section 5.4.1). Also find the indices i, j for the array $A_{GL(1)}$ such that $A_{GL(1)}[i]$ is the smallest value $\ge c$ and $A_{GL(1)}[j]$ is the largest value $\le d$.
3. Let $A_{GL(1)}[i] = y_1$ and $A_{GL(1)}[j] = y_2$. Form a rectangle $[x_1, x_4] \times [y_1, y_2]$.
4. Next, find the least common ancestor $\phi \in T_{GL(1),y}$ for the leaves storing the values y_1, y_2. Let the key stored in ϕ be y_m.
5. We split the rectangle into two smaller rectangles namely $R_1 = [x_1, x_4] \times [y_m, y_2]$ and $R_2 = [x_1, x_4] \times [y_1, y_m]$. We then count the number of maximal points in $[x_1, x_4] \times [y_1, y_2]$ by using the technique of Subproblem 3. Remember that in preprocessing phase 2, we have constructed an instance of the data structure of Theorem 7 at the node $GL(1)$. Store the count in a variable $maximalcount_{GL(1)}$.
6. Next consider the tuple $Tuple_{R_1}$ for the rectangle R_1 (see step 1f of preprocessing phase 2). Let the $count \ne 0$ in $Tuple_{R_1}$. Then, consider the y-coordinate $p_{top}(y)$ for the point p_{top} stored in the tuple and move to the next group leader $GL(2)$.
7. Consider the tuple $Tuple_{GL(2)}$ as constructed in step 5 in Section 5.4.1 and consider the range $[x'_1, x'_4]$. Also find the indices i, j for the array $A_{GL(2)}$ such that $A_{GL(2)}[i]$ is the smallest value $\ge p_{top}(y)$ and $A_{GL(2)}[j]$ is the largest value $\le d$.
8. Let $A_{GL(2)}[i] = y'_1$ and $A_{GL(2)}[j] = y'_2$. Then, for $GL(2)$, our query rectangle is $[x'_1, x'_4] \times [y'_1, y'_2]$.
9. We repeat similar steps until we visit all the group leaders. Let the number of group leader nodes be denoted by m. We then return $maximal\_count = \sum_{i=1}^{m} maximalcount_{GL(i)}$.
10. However, if the $count = 0$ in $Tuple_{R_1}$ for the step above, we then consider the $Tuple_{R_2}$ and take the topmost point in the tuple as p_{top}.

8. Query time analysis and correctness proof

Lemma 37. *Our query algorithm correctly counts the number of maximal points inside the query rectangle.*

Proof. As evident from the query algorithm, we divide the problem of reporting maximal points for a query rectangle in a general setting to $O(\frac{\log n}{\log\log n})$ instances of the problem of counting maximal points for a query rectangle in a restricted setting (Subproblem 3). By Lemma 34, it is ensured that for each such small instance, we have the correct count of maximal points. Next, notice that for a group leader $GL(i)$, we consider only the points that are above the topmost maximal point reported from the group leader $GL(i-1)$ where $GL(i-1)$ is the last visited group leader node. Thus, by summing the counts of maximal points for each of the $O(\frac{\log n}{\log\log n})$ instances, we get the count of maximal points in the query rectangle. $\square$

Lemma 38. *The query algorithm takes $O(\frac{\log n}{\log\log n})$ time to count the number of maximal points in the query rectangle.*

As the above lemma is obvious from the above discussion, we skip a detailed proof for it.

Special note. The storage space of the above discussed data structure can be improved without sacrificing the query time efficiency simply by reducing the degree of each internal node of the tree to $\log^{\rho} n$ for $0 < \rho < \frac{1}{2}$. The storage space of the above data structure will thus reduce to $O(n\frac{\log^{2+2\rho} n}{\log\log n})$.

Combining Lemma 36, Lemma 37 and Lemma 38, we conclude to Theorem 2.

References

[1] G.S. Brodal, P. Davoodi, S.S. Rao, On space efficient two dimensional range minimum data structures, Algorithmica 63 (4) (2012) 815–830.
[2] G.S. Brodal, K.G. Larsen, Optimal planar orthogonal skyline counting queries, CoRR arXiv:1304.7959, 2013.
[3] G.S. Brodal, K. Tsakalidis, Dynamic planar range maxima queries, in: ICALP (1), 2011, pp. 256–267.
[4] C.Y. Chan, H.V. Jagadish, K.-L. Tan, A.K.H. Tung, Z. Zhang, Finding k-dominant skylines in high dimensional space, in: SIGMOD Conference, 2006, pp. 503–514.
[5] D.R. Clark, J.I. Munro, Efficient suffix trees on secondary storage (extended abstract), in: SODA, 1996, pp. 383–391.
[6] A.S. Das, P. Gupta, A.K. Kalavagattu, J. Agarwal, K. Srinathan, K. Kothapalli, Range aggregate maximal points in the plane, in: WALCOM, 2012, pp. 52–63.
[7] A.S. Das, P. Gupta, K. Srinathan, Counting maximal points in a query orthogonal rectangle, in: WALCOM, 2013, pp. 65–76.
[8] J. Fischer, Optimal succinctness for range minimum queries, in: LATIN, 2010, pp. 158–169.
[9] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[10] A.K. Kalavagattu, J. Agarwal, A.S. Das, K. Kothapalli, On counting range maxima points in plane, in: IWOCA, 2012, pp. 263–273.
[11] A.K. Kalavagattu, A.S. Das, K. Kothapalli, K. Srinathan, On finding skyline points for range queries in plane, in: CCCG, 2011.
[12] C. Kejlberg-Rasmussen, Y. Tao, K. Tsakalidis, K. Tsichlas, J. Yoon, I/o-efficient planar range skyline and attrition priority queues, in: Proceedings of the 32nd ACM SIGMOD–SIGACT–SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA, June 22–27, 2013, pp. 103–114.
[13] Y. Nekrich, A linear space data structure for orthogonal range reporting and emptiness queries, Int. J. Comput. Geom. Appl. 19 (1) (2009) 1–15.
[14] Y. Nekrich, G. Navarro, Sorted range reporting, in: SWAT, 2012, pp. 271–282.
[15] F.P. Preparata, M.I. Shamos, Computational Geometry – An Introduction, Springer, 1985.
[16] P. van Emde Boas, Preserving order in a forest in less than logarithmic time, in: FOCS, 1975, pp. 75–84.
[17] C.C. Yu, W.K. Hon, B.F. Wang, Improved data structures for the orthogonal range successor problem, Comput. Geom. 44 (3) (2011) 148–159.
[18] H. Yuan, M.J. Atallah, Data structures for range minimum queries in multidimensional arrays, in: SODA, 2010, pp. 150–160.