Orthogonal Range Skyline Counting Queries
Master Thesis
Spring 2014
Computer Science - Aarhus University

Daniel Winther Petersen (20082246, dapet88@cs.au.dk)

June 19, 2014

Contents

Abstract

The Orthogonal Range Skyline of a set of points S and with a Orthogonal Range defined by p_1 and p_2 where p_2 dominates p_1, is the maximal subset of points $S' \subseteq S$ that does not contain a dominated point and where it holds for every point $p \in S'$ that $p_1.x < p.x < p_2.x$ and $p_1.y < p.y < p_2.y$. The topic of this thesis is to implement the structure from [Brodal and Larsen, 2014] over several steps, where the first steps are very simple and ending up with the final structure. The purpose is to test the performance of each step compared to each other.

The paper concludes through tests that the Naive, RMQ simple and Fractional algorithms follow the query times, preprocessing times and structure size boundaries set in the theories. There is also an analysis of the best case scenario where RMQ simple performed better than Fractional, even though Fractional where better in the worst case. This got followed by a best to worst case scenario test that showed RMQ simple quickly became worse than Fractional. At the end there were a random test that showed RMQ simple in many random cases were slower than Fractional, because of the high risk of querying a big subtree in the binary structure.

1 Introduction

Figure 1 shows an example of a Skyline (Definition 1.2) and Figure 2 shows an example of a Orthogonal range Skyline (Definition 1.3). Skyline counting queries return the number of points on the skyline and Skyline reporting queries return the points on the skyline.

Definition 1.1. *Point (x, y) dominates point (x', y') if $x \geq x'$ and $y \geq y'$.*

Definition 1.2. *The Skyline of a set of points S, is the maximal subset of points $S' \subseteq S$ that does not contain a dominated point.*

Definition 1.3. *The Orthogonal Range Skyline of a set of points S with an Orthogonal Range defined by points p_1 and p_2, where p_2 dominates p_1, is the maximal subset of points $S' \subseteq S$ that does not contain a dominated point, with respect to points in S', and where it holds for every point $p \in S'$ that $p_1.x < p.x < p_2.x$ and $p_1.y < p.y < p_2.y$.*

The data structure from [Brodal and Larsen, 2014], is a delta tree structure making use of succinct range maximum structure [Fischer, 2008], succinct dominating prefix sum structure [Raman et al., 2007] and succinct fractional successor/predecessor structures [Brodal and Larsen, 2014] to achieve orthogonal range skyline counting query time of $O(\log n / \log \log n)$ with a space usage on $O(n)$ words and $O(n \log n)$ preprocessing time.

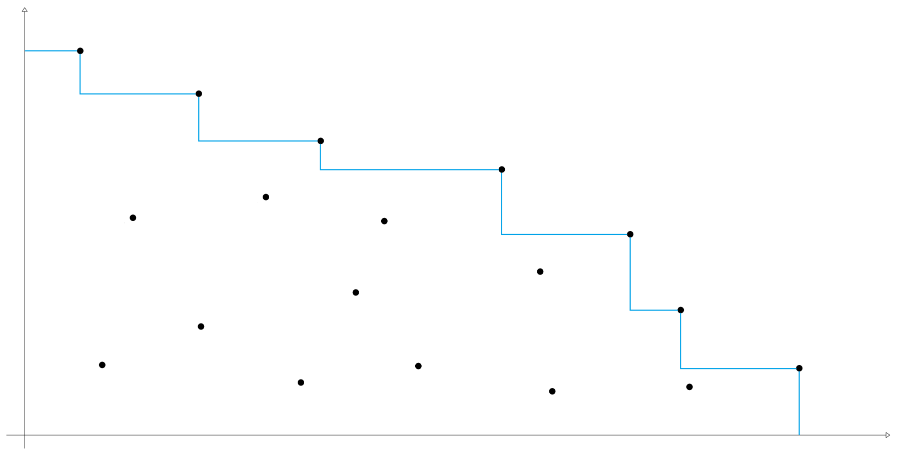
Figure 1: The figure shows the Skyline, with an unbroken blue line, for all the points.

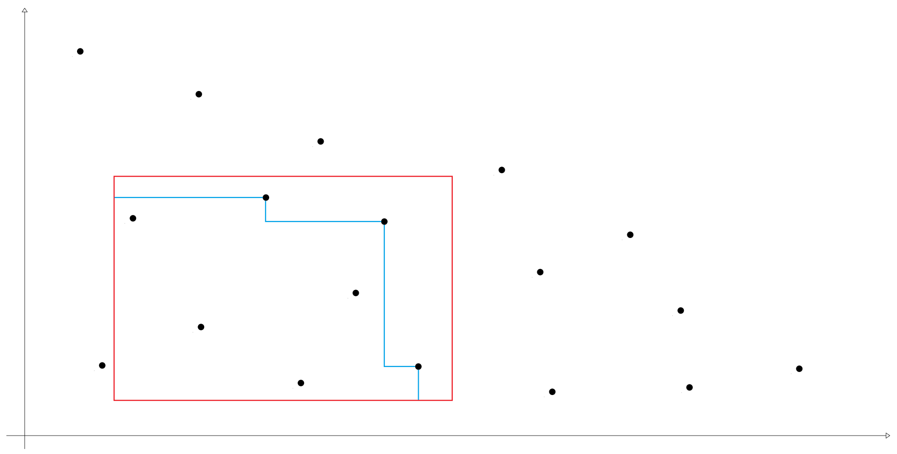
Figure 2: The figure shows the Orthogonal range Skyline, with an unbroken blue line, for all the points in the query range. The query range is represented by a red rectangle.

The topic of this thesis is to implement the structure over several steps, where the first steps are very simple and ending up with the final structure. Some steps implement a simple version of a substructure from the final structure, an example could be a range minimum structure that is not succinct, for then in a later step to implement a succinct version of the substructure. Every step will then be thoroughly tested and analyzed before proceeding to the next step. This will make it possible to compare the different steps performance and conclude which step had the best practical performance gain.

The steps with their theoretical performance:

1. Naive: $O(n)$ query time, $O(n \log n)$ preprocessing and $O(n)$ words space usage. Iteration over a sorted list.

2. RMQ simple: $O(\log^2 n)$ query time, $O(n \log^2 n)$ preprocessing and words space usage. Introducing a binary tree structure and non succinct range maximum structure [Bender and Farach-Colton, 2000] and a variation on non succinct dominating prefix sum [Brodal and Larsen, 2014] called non succinct prefix skyline count.

3. Fractional: $O(\log n)$ query time, $O(n \log^2 n)$ preprocessing and words space usage. Introducing a non succinct fractional cascading predecessor and successor structure [Chazelle and Guibas, 1986].

4. Succinct RMQ: $O(\log n)$ query time, $O(n \log n)$ preprocessing and words space usage. Introducing a succinct range maximum structure [Fischer, 2008].

5. Succinct dominating prefix sum and fractional cascading: $O(\log n)$ query time, $O(n \log n)$ preprocessing and $O(n)$ words space usage. Introducing a succinct dominating prefix sum structure and a succinct fractional cascading structure, using [Raman et al., 2007].

6. Final Structure: $O(\log n / \log \log n)$ query time, $O(n \log n)$ preprocessing and $O(n)$ words space usage. Introducing a delta tree structure [Brodal and Larsen, 2014] [Das et al., 2013] and fractional predecessor and successor structure [Brodal and Larsen, 2014].

The purpose of this paper is to test the performance of each step compared to each other.

This paper is structured with a theory section for each step and then ending with the test results and a conclusion. During the paper the first 3 steps got implemented and tested.

Definition 1.4. *Through this paper there will be multiple uses of sorting a set of points with respect to the y axis, in cases of 2 points with equal y then the point with lesser x is considered lower.*

Definition 1.5. *Through this paper there will be multiple uses of sorting a set of points with respect to the x axis, in cases of 2 points with equal x then the point with lesser y is considered lower.*

2 Previous work

All the structures mentioned here are static structures.

In [Patrascu, 2007] it was proved that a space usage of $n \log^{O(1)} n$ implies an orthogonal range counting query time of $\Omega(\log n / \log\log n)$ and in [JáJá et al., 2004] a range counting query time of $O(\log n / \log\log n)$ was achieved with a space usage on $O(n)$. Where a orthogonal range counting query reports the total amount of points within an orthogonal range of a set of points in the plane.

Orthogonal range skyline counting queries was first considered in [Das et al., 2012] where a data structure were presented with $O(n \log^2 n / \log\log n)$ space usage and a query time of $O(\log^{3/2} n / \log\log n)$. This was improved by [Kalavagattu et al., 2012] with a space usage of $O(n \log n)$ and a query time of $O(\log n)$. After that [Das et al., 2013] improved the query time to $O(\log n / \log\log n)$ but with a space usage of $O(n \log^3 n / \log\log n)$. Finally there is [Brodal and Larsen, 2014] that showed a structure that achieved a query time of $O(\log n / \log\log n)$ but with $O(n)$ space usage. This also matches the theoretical lower bound of orthogonal range skyline counting queries, also proved in [Brodal and Larsen, 2014]. The bound states that if the word size is $\log^{O(1)} n$ and the space usage is $n \log^{O(1)} n$ words then the query lower bound is $\Omega(\log n / \log\log n)$.

3 Naive

This first step is a naive brute force way of solving the problem with a query time of $O(n)$, space consumption of $O(n)$ words and a preprocessing of $O(n \log n)$. This step does not evolve later into the final algorithm and was only implemented because of its simplicity so we easily could verify its correctness. It was mainly used to debug the other steps and provide a baseline for all the tests.

The preprocessing consists of sorting all the points according to the x axis (According to Definition 1.5) and save the sorted list of points in an array A. The query can then be performed as shown in the "naiveQuery" algorithm.

First the algorithm finds all the points in the sorted array A that is within the x range of the query. This is done with a successor and predecessor search with the x query values on A. Then the algorithm iterates over all the points from high x to low x. Through the iteration there is kept track of the highest y values encountered, this value starts at the query range minimum y value. Every point encountered that have a higher y value than the highest y value previous encountered and still a lower y value than the query range maximum y value, will increase the *skylinecount* with one. After the iteration the *skylinecount* is returned.

Algorithm 1 naiveQuery(Point* * A, Point* *lowerLeft*, Point* *topRight*)

```
if $|A| == 0$ then
    return 0
$lowxIndex \leftarrow searchSucc(A, lowerLeft)$
$highxIndex \leftarrow searchPred(A, topRight)$
$lowY \leftarrow lowleft.y$
$skylineCount \leftarrow 0$
for $p \leftarrow highxIndex$ down to $lowxIndex$ do
    if $p.y > lowY$ and $p.y < topRight.y$ then
        $lowY = p.y$
        $skylineCount{+}{+}$
return $skylineCount$
```

To prove the correctness of this algorithm we need the following lemmas:

Lemma 3.1. *Let p be a point on the skyline. Then we know that all other points on the skyline with a lower x coordinate than p will also have a higher y coordinate compared to p.*

Proof. If p_1 is on the skyline and a point p_2 has a lower x and y coordinate than p_1, then we know per Definitions 1.1 and 1.2 that p_2 cannot be on the skyline because it is dominated by p_1. $\square$

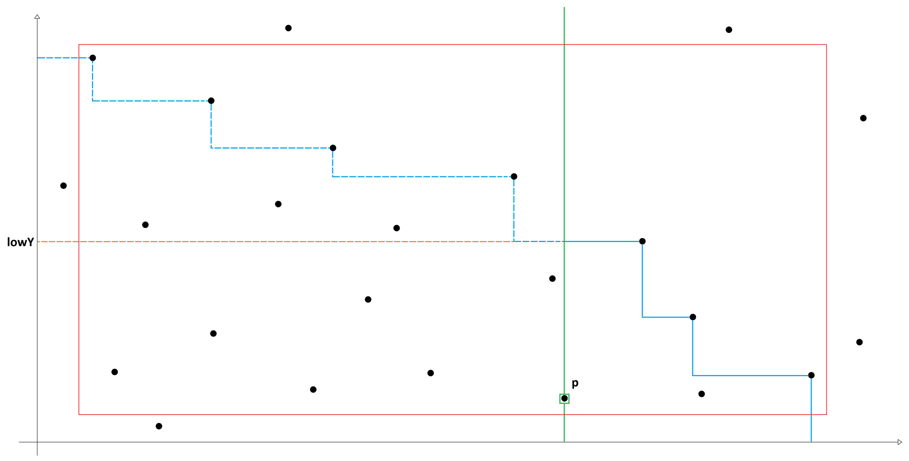

Figure 3: The figure shows a state of the Naive query during the iteration. The naive query iterates over all the sorted points from high x to low x. The green vertical line shows the current point being iterated, the blue non broken line shows the current found skyline and the broken blue line represents the skyline that will be found in future iterations. At any point through the iterations the highest y value encountered so far, $lowY$, is stored and represented by the broken orange horizontal line. This is important because the next point found with a higher y coordinate, than any other point iterated over yet, and still within the query region, shown as a red box, will be on the skyline.

Lemma 3.2. *There cannot be more points on the skyline than points in the Query set.*

Lemma 3.3. *There exists a structure that calculates the answer to an orthogonal range skyline counting query in $O(n)$ time, using $O(n)$ space and having a preprocessing time of $O(n \log n)$.*

Proof. The naiveQuery algorithm calculates the orthogonal range skyline counting query. Because of Lemma 3.2 we can justify the first **if** statement. Because the **for** loop goes from the points with highest x and down to the points with the lowest x and because of Lemma 3.1 we can conclude that the query algorithm will find every point on the skyline and no other point will be added.

The preprocessing time is equal to the time it takes to sort the array, $O(n \log n)$. The space consumption is dominated by the space for the sorted array, $O(n)$ words. The query's running time is worst case $O(n)$ because we could iterate over all the points in the array.

This structure can also return the actual points without loss of performance. □

Figure 3 shows how the state of one iteration in the Naive query could be.

4 RMQ simple

This is the second step in the implementation of the algorithm. It contains the binary tree structure, a prefix structure [Brodal and Larsen, 2014] and a simple range maximum query (RMQ) structure [Bender and Farach-Colton, 2000]. This will result in a preprocessing of $O(n \log^2 n)$, space consumption of $O(n \log^2 n)$ words and a query time of $O(\log^2 n)$.

The structure will be explained in a top down format, starting with the binary structure and go on to the lower structures afterwards.

4.1 Binary structure

A query in a binary search tree over all the points, in this case sorted on their x coordinates, can still find the minimum and maximum point relative to the x-axis of the query range in $O(\log n)$ time. Another property is that the paths from those two points to their least common ancestor (LCA) each have up to $O(\log n)$ subtrees within the x range of the query.

All points contained in these subtrees are within the query range relative to the x-axis, but nothing is known about their y coordinates yet. It is this fact that the rest of this algorithm builds upon. Because the query will again traverse from high x to low x, starting in the lowest point in the path with high x and go up until the LCA, then down from there through the low x path.

Figure 4 shows the binary structure over some random points, and how one query could look like with *Xmin*, *Xmax*, high x path and the low x path.

The following lemma will be needed:

Lemma 4.1. *If p is the point within the query with the largest x coordinate, then p is on the skyline.*

Proof. If p is the point with the highest x coordinate then there cannot exist another point within the query range that could dominate p, so p must be on the skyline. □

The query can easily deal with the two leaf cases, *Xmin* and *Xmax*. In the *Xmax* case if it is in the y range of the query then it is the first point in the skyline (Lemma 4.1). If the query keeps track of the highest y coordinate of previous added points, as in the naive case, then the query can check if *Xmin* is in the range. How the query handle the subtree cases is explained in the next two sections. The pseudo code "rmqSimpelQuery" shows in more detail how the query is implemented.

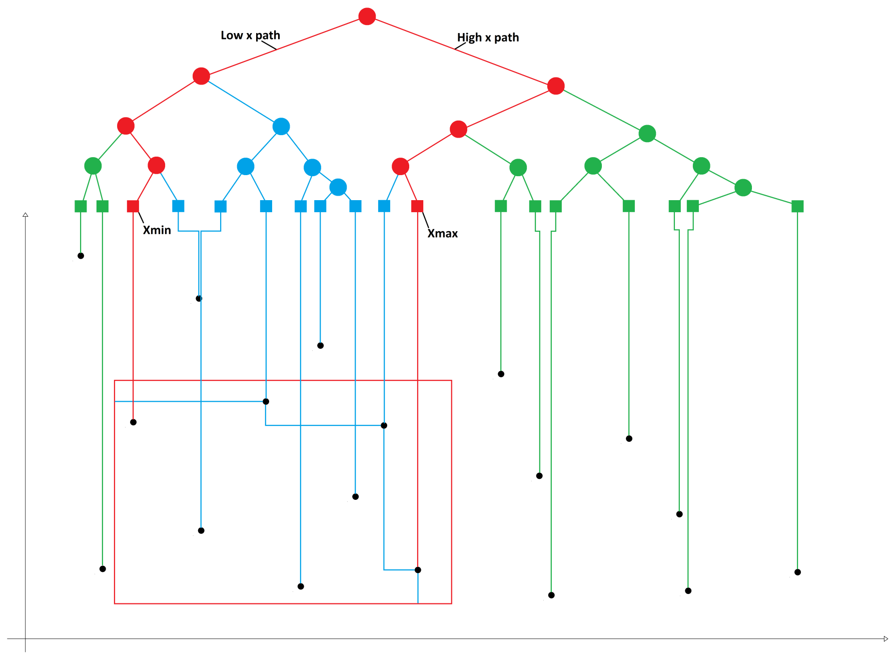

Figure 4: The figure shows how the binary structure in the RMQ Simpel structure is used in the query. The query region is shown as a red box. Based on the query regions x values the query will find $Xmin$ and $Xmax$ by a predecessor and successor search through the tree. At the same time the high x path and the low x path are being calculated, represented as red nodes and edges. All leafs and inner nodes that are within the query region x values and is not a part of the high or low x paths is marked in blue, while the rest are marked in green. It is worth to note that not every leaf has the same path length to the root and that any point within the valid section of the binary tree, including $Xmin$ and $Xmax$, not necessarily is within the y range of the query range but that they are within the x range. What is left is to deal with the leaf cases and all the subtrees of the valid section.

Algorithm 2 rmqSimpelQuery(Point *lowerLeft*, Point *topRight*)

```
$highPath \leftarrow rootNode.getPredecessorPath(topRight.x)$
$lowPath \leftarrow rootNode.getSuccessorPath(lowerLeft.x)$
$lca \leftarrow findLCA(highPath, lowPath)$
$lowY \leftarrow lowerLeft.y$
$highY \leftarrow topRight.y$
if $highPath[lca] = lowPath[lca]$ then
    return $subQuery(highPath[lca], lowY, highY)$
$skylineCount \leftarrow 0$
for $i \leftarrow |highPath| - 1$ to $lca$ do
    $leftChild \leftarrow highPath[i].leftChild$
    if $leftChild = null$ then
        $skylineCountChange \leftarrow highPath[i].rmqSubQuery(lowY, highY)$
        if $skylineCountChange \neq 0$ then
            $lowY \leftarrow highPath[i].predecessor(highY).y$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
    elseif $leftChild \neq highPath[i+1]$ then
        $skylineCountChange \leftarrow leftChild.rmqSubQuery(lowY, highY)$
        if $skylineCountChange \neq 0$ then
            $lowY \leftarrow leftChild.predecessor(highY).y$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
for $i \leftarrow lca - 1$ to 0 do
    $rightChild \leftarrow lowPath[i].rightChild$
    if $rightChild = null$ then
        $skylineCountChange \leftarrow lowPath[i].rmqSubQuery(lowY, highY)$
        if $skylineCountChange \neq 0$ then
            $lowY \leftarrow lowPath[i].predecessor(highY).y$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
    elseif $rightChild \neq lowPath[i+1]$ then
        $skylineCountChange \leftarrow rightChild.rmqSubQuery(lowY, highY)$
        if $skylineCountChange \neq 0$ then
            $lowY \leftarrow rightChild.predecessor(highY).y$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
return $skylineCount$
```

4.2 Prefix skyline count structure

With the indexes of the points on the skyline with the highest and lowest y coordinate, then the skyline count over all the points can be computed in $O(1)$ time by using the prfix skyline count.

Definition 4.2. *Prefix skyline count is the skyline count for a given point in a set, if you only consider the points with lower y coordinates than that point in the set.*

Algorithm 3 generatingPrefixSkylineCount(Point* *A)

```
$prefixSkylineCount \leftarrow int[|A|]$
$nonDom \leftarrow \{\}$
for $p \leftarrow 0$ to $|A| - 1$ do
    while $nonDom \neq \phi$ AND $nonDom.peak().x \leq A[p].x$ do
        $nonDom.pop()$
    end while
    $nonDom.push(p)$
    $prefixSkylineCount[p] \leftarrow |nonDom|$
```

Lemma 4.3. *The algorithm "generatingPrefixSkylineCount" will generate an array of prefix skyline counts, one for each point in the input y sorted array A, in $O(n)$ time and using $O(n)$ space.*

Proof. Before each iteration of the for loop the stack *nonDom* contains the skyline of $A[0..p-1]$, based on the Definition 1.2. The point $A[p]$ will have a higher y coordinate than all points in *nonDom*. Because *nonDom* contains the skyline of the previous points in a increasing y order and decreasing x order, then the top point will have the lowest x coordinate, else it would dominate some of the lower points and then *nonDom* will not be a true skyline. So the while loop starts from the top of the stack and stops at the first point that haves a higher x coordinate, because the stack is sorted with respect to x. Every point will be added to the *nonDom* and only be removed if it was dominated, and *nonDom* is a true skyline at the end of every iteration, so we can conclude that we will get a correct prefix skyline count.

Every point will be added to *nonDom* only once and there will at maximum be done $2n$ comparisons on the points in *nonDom*. Because beside the first check in the while loop at every iteration, then every other check will remove a point from *nonDom* and there can only be removed $n - 1$ points through the whole iteration. This sums up to a running time of $O(n)$ and the *prefixSkylineCount* array of n size is the only thing that needs to be saved so that limits the space usage to $O(n)$. □

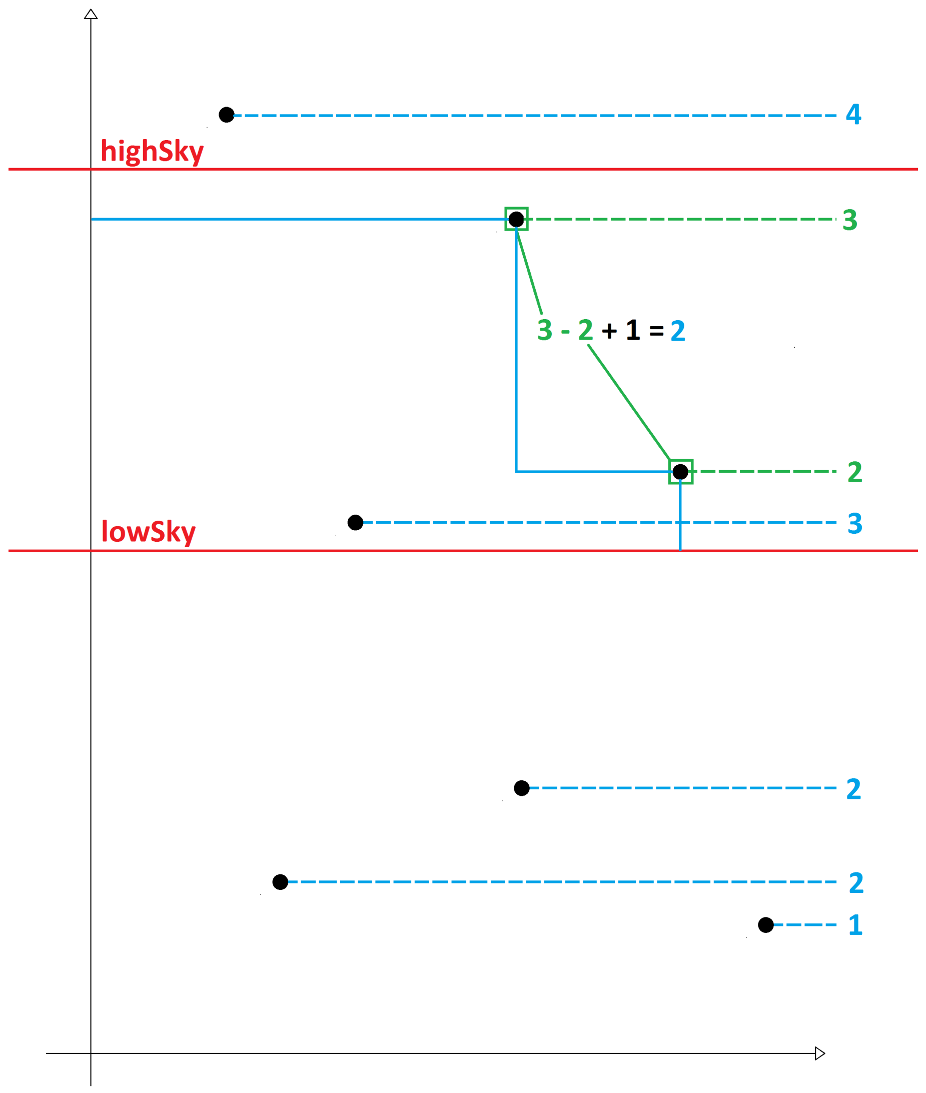

Figure 5: Every inner node in the binary structure, in the RMQ simple structure, contains a prefix skyline count structure, over all the points in its subtree. The figure shows all the points in a subtree and pictures how one query could be. The blue broken horizontal lines link a point with its prefix sum count. The two red lines show the query region, because every point of a queried subtree is within the x range then only the y range is shown. The blue unbroken line shows the Skyline within the query range. The two green squares mark the $highSky$ and $lowSky$ points, and in the middle is shown the $prefixSkylineCount[highSky] - prefixSkylineCount[lowSky] + 1$ calculation based on those points. $highSky$ will be found by a predecessor search on the points and $lowSky$ will be found through the RMQ structure that is explained next.

Lemma 4.4. *With the indexes of the points on the skyline with the highest and lowest y coordinate, then the skyline count over all the points can be computed by:*
$\mathit{prefixSkylineCount}[\mathit{highSky}] - \mathit{prefixSkylineCount}[\mathit{lowSky}] + 1$ *in* $O(1)$ *time.*

Proof. $\mathit{prefixSkylineCount}[\mathit{highSky}]$ is the total skyline count from highest point down to the lowest point in the total set, not just in the query range, and $\mathit{prefixSkylineCount}[\mathit{lowSky}]$ is the part we do not need to consider. The plus one is because $\mathit{prefixSkylineCount}[\mathit{lowSky}]$ includes the *lowSky* that is within the query range. So all non dominated points between *highSky* and *lowSky*, both included, with a constant time lookup is left. □

Figure 5 shows the Prefix skyline count structure over some random points and how one query could make use of it to find the skyline count between two points.

4.3 RMQ structure

With a range maximum structure the indexes of the point on the skyline with the lowest y coordinate can be found in $O(1)$ time, if the predecessor to $highY$ and successor to $lowY$ is known.

Lemma 4.5. *There exists a structure that can find the highest point on the x axis in a set of points S sorted on y axis, in a range on the y axis, with query time $O(1)$, space usage $O(n \log n)$ and preprocessing time $O(n \log n)$.*

Proof. Any interval in S, $S[i, j]$ can be covered by two smaller intervals $S[i, i + k]$ and $S[j - k, j]$, where $k = 2^{\lfloor \log |S| \rfloor}$, without covering more than S. Worst case the 2 intervals need to be $2^{\lfloor \log n \rfloor}$ size, because the range can be all the points. With this upper bound all possible intervals on S can be covered by using $\lfloor \log n \rfloor + 1$ indexes for each input point.

For each of the $\log n$ indexes the point with the largest x coordinate in its range is saved. Then the point with the highest x coordinate in any interval can be calculated in $O(1)$ time, by comparing the two intervals max points. This structure will have $O(n \log n)$ space usage.

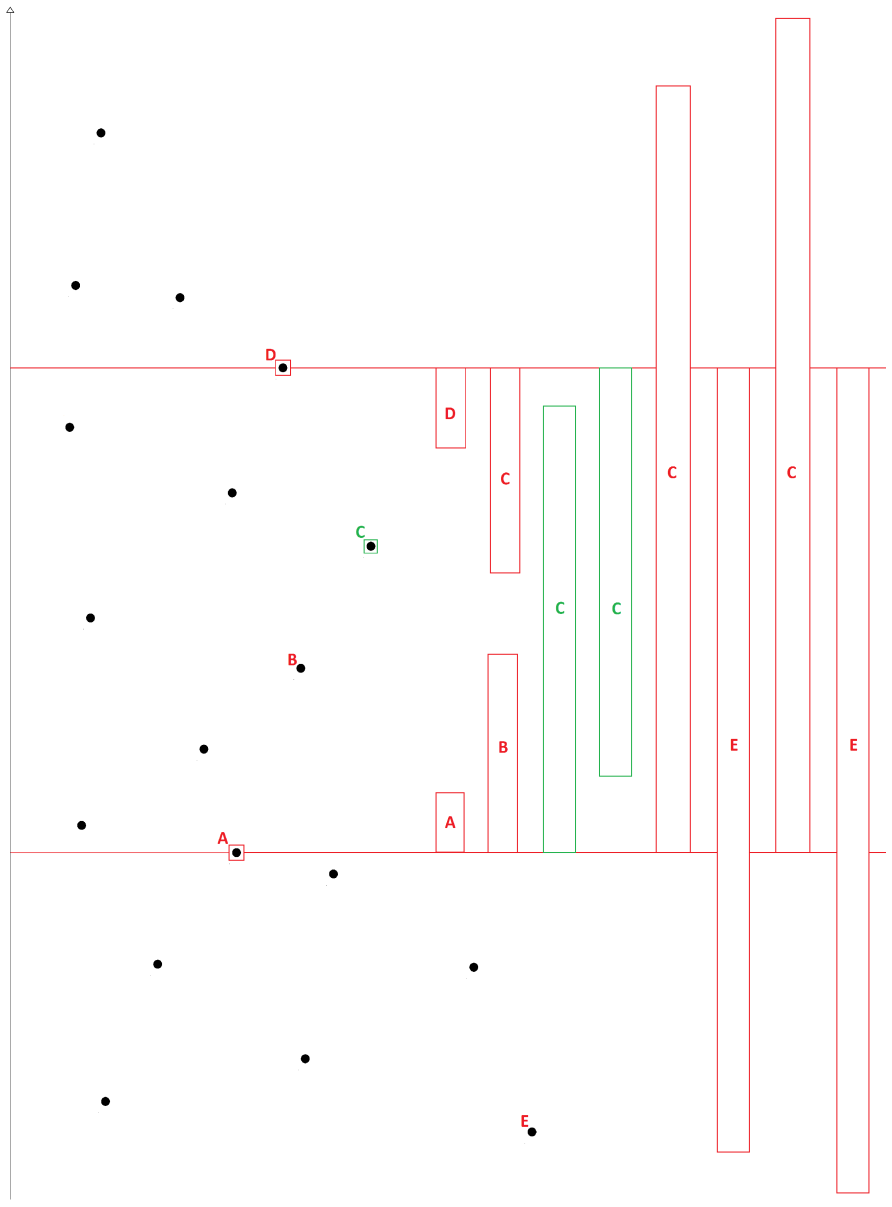

Figure 6: Every inner node in the binary structure, of the RMQ Simple structure, will have a RMQ structure over all the points in its subtree. The figure shows what is stored for two points in the RMQ structure and how a query uses the structure to find *lowSky*. There is stored $\lfloor \log n \rfloor = \lfloor \log 18 \rfloor = 4$ points for each point in the subtree. The two last indexes at both points covers the same area, but this will not be the case if the points had been further away from the center point. During the query the two points marked A and D is found with a predecessor or successor search on the queries y range. Then the distance between the two points will determine the index to use $\lfloor \log mainIntervalSize \rfloor = \lfloor \log 9 \rfloor = 3$ at each point to find the point with the largest x value. The rectangles representing these indexes are colored green.

The fact that any index $S[i,j]$ can be covered by two smaller intervals $S[i,i+k]$ and $S[j-k,j]$ of $k = 2^{\lfloor \log |S| \rfloor}$ size means that every index in the RMQ structure can be calculated in O(1) time, because $RMQ[level, index] = \max(RMQ[level-1, index], RMQ[level-1, index+2^{level-1}])$. If the structure is filled bottom up, starting from the indexes representing the shortest intervals and then going up to the bigger ones, as shown in the pseudo code *generateRMQ* that takes a list of points, sorted on the y axis, then there exist $n \log n$ indexes that all can be filled out in $O(1)$ time and this totals up to a preprocessing time of $O(n \log n)$. □

Figure 6 shows the RMQ structure over some random points and how a query could use it to find the lowest point on the skyline.

Algorithm 4 generateRMQ(Point* *A)

```
$rmqForward \leftarrow int[\log|A|][|A|]$
for $i \leftarrow 0$ to $|A|$ do
    $rmqForward[0][i] \leftarrow i$
for $layer \leftarrow 1$ to $\log|A|$ do
    for $i \leftarrow 0$ to $|A|-1$ do
        $leftSubIndex \leftarrow i$
        $rightSubIndex \leftarrow i + 2^{layer-1}$
        if $rightSubIndex \geq |A|$ then
            $rightSubIndex \leftarrow |A|-1$
        $leftSubRMQ \leftarrow rmqForward[layer-1][leftSubIndex]$
        $rightSubRMQ \leftarrow rmqForward[layer-1][rightSubIndex]$
        if $A[leftSubRMQ].x > A[rightSubRMQ].x$ then
            $rmqForward[layer][i] \leftarrow leftSubRMQ$
        else
            $rmqForward[layer][i] \leftarrow rightSubRMQ$
```

4.4 Conclusion

Lemma 4.6. *The point p within the query range with the highest y value is the highest point on the skyline.*

Proof. Because there is no point within the query range that has higher y value, then the point p cannot be dominated. □

Lemma 4.7. *There exists a structure that can perform an orthogonal range skyline count query in $O(\log^2 n)$ time, that uses $O(n \log^2 n)$ space and has a preprocessing time of $O(n \log^2 n)$*

Proof. There are two paths in the binary structure, with up to $\log n$ subtrees to check in each paths. For each subtree the query can find the top point within the query range in $O(\log n)$ time, with a binary predecessor search. The lowest point within the query range that is above the highest y coordinate encountered so far to the right of the subtree, through the previous iterations through the subtrees, can be found with a binary successor search in $O(\log n)$ time. The highest point in the query range is equal to the highest point on the skyline by Lemma 4.6 and the lowest point on the skyline can be found with the RMQ structure from Lemma 4.5 in $O(1)$ time. Then the skyline count can be calculated in $O(1)$ time through the prefix skyline count structure by Lemma 4.4. This will total up to $O(\log^2 n)$ time for a query.

The binary structure has $\log n$ layers, every element will have $O(\log n)$ indexes in one of the RMQ structures found on a layer. Every element will also be represented in the dominating prefix sum structure once in each layer. This totals up to $O(n \log n \log n + n) = O(n \log^2 n)$ space.

The preprocessing uses the same argument: $\log n$ layers that spans n elements each. For every inner layer that spans n elements there will be used $\log n$ time in the RMQ preprocessing and $O(1)$ time in the prefix skyline count per element. The binary tree can be created in $O(n)$ time bottom up because the input is sorted. This totals up to a preprocessing time of $O(n + n \log n \log n) = O(n \log^2 n)$. $\square$

The pseudo code "rmqSubQuery" shows how the sub query into every subtree is implemented.

Algorithm 5 rmqSubQuery(int *lowY*, int *highY*)

```
$inHighY \leftarrow predecessor(pointsSortY, highY)$
$inLowY \leftarrow successor(pointsSortY, lowY)$
**if** $inLowY = -1$ OR $inHighY = -1$ OR $inHighY < inLowY$ **then**
    return 0
**if** $innerNode$ **AND** $inHighY > inLowY$ **then**
    $highLowDist \leftarrow inHighY - inLowY$
    $logDist \leftarrow \lfloor \log highLowDist \rfloor$
    $lowest \leftarrow rmq[logDist][lowY]$
    $highest \leftarrow rmq[logDist][highY - 2^{logDist}]$
    **if** $pointsSortY[lowest].x > pointsSortY[highest].x$ **then**
        $lowestMemberOfSkyline \leftarrow lowest$
    **else**
        $lowestMemberOfSkyline \leftarrow highest$
    **return** $dps[inHighY] - dps[lowestMemberOfSkyline] + 1$
**return** 1
```

5 Fractional cascading

This is the third step in the implementation of the algorithm. It contains the non succinct fractional cascading structures, these are a simple variation of the fractional cascading from [Chazelle and Guibas, 1986]. This will result in a preprocessing time of $O(n \log^2 n)$, space consumption of $O(n \log^2 n)$ words and a query time of $O(\log n)$. First there will be some Lemmas stating the fractional cascading substructures that will be used and then there will be two big Lemmas explaining the implementation of the structures in high and low x path.

Lemma 5.1. *If there exists two ordered lists of points sorted on the y coordinate, S_1 and S_2, with size $O(n)$, then there exists a Fractional Cascading structure that supports minimum successor queries on y, that for any point $p_1 \in S_1$ returns the point $p_2 \in S_2$ with the lowest y value where $p_1.y < p_2.y$, with a query time of $O(1)$, preprocessing in $O(n)$ time and space usage in $O(n)$ words.*

Algorithm 6 fracSuccessor(S_1, S_2)

```
$s2Counter \leftarrow 0$
$fracCascSucc \leftarrow int[|S_1|]$
for $i \leftarrow 0$ to $|S_1| - 1$ do
    while $s2Counter \neq -1$ AND $S_1[i].y \leq S_2[s2Counter].y$ do
        if $s2Counter < |S_2| - 1$ then
            $s2Counter++$
        else
            $s2Counter \leftarrow -1$
            break
    end while
    $fracCascSucc[i] \leftarrow s2Counter$
return $fracCascSucc$
```

Proof. The pseudo code "fracSuccessor" shows how the preprocessing could be done within $O(n)$ time and words space. The code makes use of the fact that two points from S_1 with indexes i and j, where $i < j$, minimum successor indexes in S_2 will have the property $succ(i) \leq succ(j)$. This means, as shown by the while loop, that if we iterate through both lists at the same time, from low to high, then every current iterated point in S_2, p_2, that is not the minimum successor to the current iterated element in S_1, p_1, will not be a minimum successor to any point later in the iteration over S_1 either, and if p_2 is bigger than p_1 then there will be no better candidate later in S_2, because they are sorted. This way the preprocessing will only touch every element once and this result in the preprocessing time on $O(n)$ time. The space usage is bounded by the *fracCascPred* array so $O(n)$ words and the query is a $O(1)$ lookup in *fracCascPred*. □

Lemma 5.2. *The structure from Lemma 5.1 can also support, with the same performance, minimum successor on y if no equal queries, that for any point* $p_1 \in S_1$ *returns the point* $p_2 \in S_2$ *with the lowest y value where* $p_1.y \leq p_2.y$.

Proof. The same proof as for Lemma 5.1, the $S_1[i].y \leq S_2[s2Counter].y$ condition in the while loop just need to be changed to $S_1[i].y < S_2[s2Counter].y$, to only give the minimum successor with respect to y if there is no point equal to. □

Lemma 5.3. *The structure from Lemma 5.1 can also support, with the same performance, maximum predecessor on y if no equal queries, that for any point* $p_1 \in S_1$ *returns the point* $p_2 \in S_2$ *with the highest y value where* $p_1.y \geq p_2.y$.

Proof. The same proof as for Lemma 5.1, the $S_1[i].y \leq S_2[s2Counter].y$ condition in the while loop just need to be changed to $S_1[i].y > S_2[s2Counter].y$ and the directions changed from "low to high" to "high to low", to only give the maximum predecessor with respect to y if there is no element equal to. □

Lemma 5.4. *If there exists two ordered lists of points sorted on the y coordinate, child and parent, with size* $O(n)$, *and child* $\subset$ *parent then there exists a Fractional Cascading structure that supports child to parent mapping queries, that finds the points from parent that is equal to the queried points in child, with a query time of* $O(1)$, *preprocessing will take* $O(n)$ *time and space usage on* $O(n)$.

Proof. This proof is very similar to the proof for Lemma 5.1, because the points are sorted then the same facts hold, that two points from *child* with indexes i and j, where $i < j$, mapped indexes in *parent* will have the property $map(i) \leq map(j)$. Because *child* is a subset of *parent* then every *child* element does exist in the *parent* set. This means the $S_1[i].y \leq S_2[s2Counter].y$ condition from the pseudo code from Lemma 5.1 needs to be changed to $child[i].y = parent[s2Counter].y$ and then the lookup table can be constructed in $O(n)$ time. So the preprocessing time is $O(n)$, query time is equal to the lookup so $O(1)$ time and the space usage is $O(n)$ because of the table. □

Lemma 5.5. *If the sub query of RMQ simple query mentioned in Lemma 4.7 could get the index on lowY and highY, inLowY and inHighY, instead of their values as parameters, then the sub query could be performed in $O(1)$ time.*

Proof. If the sub query were given *inLowY* and *inHighY* then there were no need for the two binary searches to find the indexes. Then what is left in the query is: two lookups in the RMQ table and two lookups in the prefix skyline count table, both of these are $O(1)$ time lookups and sums up to a query time of $O(1)$. This is also shown in the pseudo code "fracSubQuery". □

Algorithm 7 fracSubQuery(int *inLowY*, int *inHighY*)

```
if $inLowY = -1$ OR $inHighY = -1$ OR $inHighY < inLowY$ then
    return 0
if $innerNode$ then
    if $inHighY > inLowY$ then
        $highLowDist \leftarrow inHighY - inLowY$
        $logDist \leftarrow \lfloor \log highLowDist \rfloor$
        $lowest \leftarrow rmq[logDist][lowY]$
        $highest \leftarrow rmq[logDist][highY - 2^{logDist}]$
        if $pointsSortY[lowest].x > pointsSortY[highest].x$ then
            $lowestMemberOfSkyline \leftarrow lowest$
        else
            $lowestMemberOfSkyline \leftarrow highest$
        return $psc[inHighY] - psc[lowestMemberOfSkyline] + 1$
return 1
```

5.1 High x path

Lemma 5.6. *It is possible for the high x path part of the RMQ simple query mentioned in Lemma 4.7 to give the indexes of* $lowY$ *and* $highY$*,* $inLowY$ *and* $inHighY$*, to every sub query during the query, resulting in a query time of* $O(\log n)$ *during this part.*

Proof. The high x path part is a bottom up iteration of the high x path, but the calculation of the path is a top down recursive call from the root down to $Xmax$. During this recursive call the $inLowY$ and $inHighY$ of every node can be calculated from the parent nodes $inLowY$ and $inHighY$ with a "maximum predecessor if no equal query" (Lemma 5.3) and a "minimum successor if no equal query" (Lemma 5.2) in $O(1)$ time. The root will make use of a successor and predecessor call on a sorted list of the input points to find its $inLowY$ and $inHighY$, so these points would be points from the input set that is within the query range, with respect to the x coordinate. The "if no equal queries" needs to be used, else every recursion will exclude points that is within the query range, with respect to the x coordinate.

The first call to the sub query, at $Xmax$, can be done in $O(1)$ based on Lemma 5.5, because the query now knows $inLowY$ and $inHighY$.

The next sub queries, during the iteration through the high x path, will be called on inner nodes that are left children to the high x path, these parent nodes already calculated its own $inLowY$ and $inHighY$ based on the query values, during the recursion. But if earlier, during the query, a point were added to the Skyline then $lowY$ needs to change in the parent nodes. There is several parts to this:

When a sub query adds to the skyline then that child's parents $inLowY$ needs to change to the parent index of the child's $inHighY$, to keep track of the highest y values added yet. This can be done in $O(1)$ time with the child to parent mapping mentioned in Lemma 5.4, $parent.lowY \leftarrow child.childToParMap(child.highY)$, because the points in the child's subtree is a subset to the parents.

When the query iterates up through the high path, if there have been added to the skyline previously then the $inLowY$ of the parent node just iterated to, $parent_1$, must be equal to the index in $parent_1$ of the element at $inLowY$ of the previously iterated parent, $parent_2$. Because $parent_2 \subset parent_1$ then by using Lemma 5.4:

$parent_1.lowY \leftarrow parent_2.childToParMap(parent_2.highY)$ can be done in $O(1)$ time.

The last part is then to call the next sub query. If there have not been added to the Skyline yet, then *inLowY* and *inHighY* of the child can be calculated with a "maximum predecessor if no equal query" (Lemma 5.3) and a "minimum successor if no equal query" (Lemma 5.2) in $O(1)$ time. But if there have been added to the Skyline previously then the *inLowY* is now representing a point that have already been added to the skyline, so the *inLowY*s y coordinate in the child may not be equal to the *inLowY*s y coordinate in the parent anymore, so to calculate *inLowY* the "minimum successor query" (Lemma 5.1) is needed, it also have a query time of O(1).

This shows that the high x path part of the RMQ simple query is able to give *inLowY* and *inHighY* to every sub query. It also shows that calculating these indexes at every node takes in total $O(1)$ time and Lemma 5.5 shows us that every sub query also only takes $O(1)$ time, if given *inLowY* and *inHighY*. With $O(\log n)$ nodes in the high x path then it sums up to a total query time of $O(\log n)$ total for the high x path part.

All these details is also shown in the first half of the pseudo code "fracQuery". □

Algorithm 8 fracQuery(Point *lowerLeft*, Point *topRight*):FirstHalf

```
$highPath \leftarrow rootNode.getPredecessorPath(topRight.x)$
$lowPath \leftarrow rootNode.getSuccessorPath(lowerLeft.x)$
$lca \leftarrow findLCA(highPath, lowPath)$
**if** $highPath[lca] = lowPath[lca]$ **then**
    **return** $subQuery(highPath[lca], lowY, highY)$
$skylineCount \leftarrow 0$
$highPathhighs \leftarrow int[|highPath|]$
$highPathlows \leftarrow int[|highPath|]$
$highs[0] \leftarrow rootNode.getPredecessor(topRight.x)$
$lows[0] \leftarrow rootNode.getSuccessor(lowerLeft.x)$
**for** $i \leftarrow 1$ **to** $|highPath| - 1$ **do**
    $highs[i] \leftarrow highPath[i].fracParPredIfNoEqual(highs[i-1])$
    $lows[i] \leftarrow highPath[i].fracParSucIfNoEqual(lows[i-1])$
**for** $i \leftarrow |highPath| - 1$ **to** $lca + 1$ **do**
    $leftChild \leftarrow highPath[i].leftChild$
    **if** $leftChild = null$ **then**
        $highY \leftarrow highs[i]$
        $lowY \leftarrow lows[i]$
        $skylineCountChange \leftarrow highPath[i].fracSubQuery(lowY, highY)$
        **if** $skylineCountChange \neq 0$ **then**
            $lows[i-1] \leftarrow highPath[i].childParMap(highY)$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
    **elseif** $leftChild \neq highPath[i+1]$ **then**
        $highY \leftarrow leftChild.fracParPredIfNoEqual(highs[i])$
        $lowY \leftarrow leftChild.fracParSucIfNoEqual(highPathlows[i])$
        **if** $skylineCount \neq 0$ **then**
            $lowY \leftarrow leftChild.fracParSuc(highPathlows[i])$
        $skylineCountChange \leftarrow leftChild.fracSubQuery(lowY, highY)$
        **if** $skylineCountChange \neq 0$ **then**
            $lows[i] \leftarrow leftChild.childParMap(highY)$
            $lows[i-1] \leftarrow highPath[i].childParMap(lows[i])$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
```

5.2 Low x path

Lemma 5.7. *It is possible for the low x path part of the RMQ simple query mentioned in Lemma 4.7 to give the indexes of $lowY$ and $highY$, $inLowY$ and $inHighY$, to every sub query during the query, resulting in a running time of $O(\log n)$ for the query during this part.*

Proof. The low x path part is a top down iteration of the low x path. This means that $inLowY$ and $inHighY$ can, and will be, calculated during the iteration and not necessarily during the recursion, that created the low x path. So every iterations parent nodes $inLowY$ and $inHighY$ will be calculated based on its parent node, as in the high x path case. In the case where there is never added to the skyline, then at every iteration the $inLowY$ and $inHighY$ of the child can be calculated with a "maximum predecessor if no equal query" (Lemma 5.3) and a "minimum successor if no equal query" (Lemma 5.2), because the previous $inLowY$ and $inHighY$ never were a part of the skyline but are within the query range. The same goes for the $inLowY$ and $inHighY$ of all the sub queries to the children of the path.

But if there have been added to the skyline in the previous iterations then the $inLowY$ of the previous iteration is part of the skyline already, so the $inLowY$ of the current iteration need to be found through a "minimum successor query" (Lemma 5.1), because the child $inLowY$ may not have the same y coordinate as the $inLowY$ in the parent. But if this iterations sub query does not add to the skyline, then the next iterations $inLowY$ will be calculated with a "minimum successor if no equal query" (Lemma 5.2), because the current iterations new $inLowY$ were not part of the skyline. This is also the case for the first iteration, the left child of the lca, and no matter if there is added to the skyline or not the queries to find the $inLowY$ for the sub queries do not change.

This shows that the low x path part of the RMQ simple query is able to give $inLowY$ and $inHighY$ to every sub query. It also shows that calculating these indexes at every node takes in total $O(1)$ time and Lemma 5.5 shows us that every sub query also only takes $O(1)$ time, if given $inLowY$ and $inHighY$. With $O(\log n)$ nodes in the low x path then it sums up to a total query time of $O(\log n)$.

All these details is also shown in the second half of the pseudo code "fracQuery". □

Algorithm 9 fracQuery(Point *lowerLeft*, Point *topRight*):SecondHalf

```
$highY \leftarrow lowPath[lca+1].fracParPredIfNoEqual(highs[lca])$
if $skylineCount > 0$ then
    $lowY \leftarrow lowPath[lca+1].fracParSuc(lows[lca])$
else
    $lowY \leftarrow lowPath[lca+1].fracParSucIfNoEqual(lows[lca])$
for $i \leftarrow lca - 1$ to $0$ do
    $foundPointInLow \leftarrow false$
    $rightChild \leftarrow lowPath[i].rightChild$
    if $rightChild = null$ then
        $skylineCountChange \leftarrow lowPath[i].fracSubQuery(lowY, highY)$
        if $skylineCountChange \neq 0$ then
            $skylineCount \leftarrow skylineCount + skylineCountChange$
    elseif $rightChild \neq lowPath[i+1]$ then
        $highYChild \leftarrow rightChild.fracParPredIfNoEqual(highY)$
        $lowYChild \leftarrow rightChild.fracParSucIfNoEqual(lowY)$
        $skylineCountChange \leftarrow rightChild.fracSubQuery(lowYChild, highYChild)$
        if $skylineCountChange \neq 0$ then
            $lowY \leftarrow rightChild.childParMap(highYChild)$
            $skylineCount \leftarrow skylineCount + skylineCountChange$
            $foundPointInLow \leftarrow true$
        $highY \leftarrow lowPath[lca+1].fracParPredIfNoEqual(parentHigh)$
        if $foundPointInLow$ then
            $lowY \leftarrow lowPath[lca+1].fracParSuc(lowY)$
        else
            $lowY \leftarrow lowPath[lca+1].fracParSucIfNoEqual(lowY)$
return $skylineCount$
```

5.3 Fractional performance

Lemma 5.8. *There exists a structure that can perform orthogonal range skyline counting queries in $O(\log n)$ time, that uses $O(n \log^2 n)$ space and has a preprocessing running time of $O(n \log^2 n)$.*

Proof. Based on the RMQ simple query mentioned in Lemma 4.7 and with the changes to the high and low x paths mentioned in Lemma 5.6 and Lemma 5.7, then the query is able to achieve the $O(\log n)$ query time total. These changes needs the fractional structures of "maximum predecessor if no equal query" (Lemma 5.3), "minimum successor if no equal query" (Lemma 5.2), "minimum successor query" (Lemma 5.1) and the child to parent mapping mentioned in Lemma 5.4, these all add a $O(n)$ preprocessing time and a $O(n)$ space usage to every node, but these are smaller than the RMQ structures $O(n \log n)$ preprocessing time and $O(n \log n)$ space usage for every node, so these aspects of the performance are unchanged resulting in a $O(n \log^2 n)$ space and a preprocessing running time of $O(n \log^2 n)$ total, with a $O(\log n)$ query time total. $\square$

6 Succinct RMQ

This is the fourth step in the implementation of the algorithm. It replaces the non succinct range maximum structure with the succinct one found in [Fischer, 2008]. This will result in a preprocessing time of $O(n \log n)$, space consumption of $O(n \log n)$ words and a query time of $O(\log n)$.

[Fischer, 2008] states that:

Lemma 6.1. *There exists a structure that can find the highest point on the x axis in a set of points S sorted on y axis, in a range on the y axis. With query time $O(1)$, space usage $O(n)$ bits and preprocessing time $O(n)$.*

Lemma 6.2. *There exists a structure that can perform an orthogonal range skyline count query in $O(\log n)$ time, that uses $O(n \log n)$ space and has a preprocessing time of $O(n \log n)$*

Proof. By replacing the non succinct RMQ structure from Lemma 4.7 with the succinct RMQ structure from Lemma 6.1, then the query time is unchanged but for every node the preprocessing and space usage is decreasing. Every point is present in $O(\log n)$ inner nodes and in every inner node there is a RMQ structure, a prefix skyline count structure and $O(1)$ fractional cascading structures, beside this by using the succinct RMQ structure there is no need for the *sortedY* lists in the inner nodes anymore. The fractional cascading structures have a space usage on $O(1)$ words and a preprocessing of $O(1)$, for each point (Lemma 5.1, 5.2, 5.3, 5.4). The prefix skyline count structure has a space usage of $O(1)$ words and a preprocessing of $O(1)$, for each point (Lemma 4.3). So with the succinct RMQ structure there is a space usage of $O(1)$ words and a preprocessing time of $O(1)$ for each point, this totals up to a preprocessing of $O(n \log n)$ and a space usage of $O(n \log n)$ words for the total structure. □

7 Succinct dominating prefix sum and fractional cascading

This is the fifth step in the implementation of the algorithm. It replaces the non succinct prefix skyline count with succinct dominating prefix sum and the non succinct fractional cascading with a succinct one, both changes uses the prefix sum structure from [Raman et al., 2007]. This will result in a preprocessing time of $O(n \log n)$, space consumption of $O(n)$ words and a query time of $O(\log n)$.

[Raman et al., 2007] states that:

Lemma 7.1. *Let $X[1..s]$ be a vector of s non-negative integers with a total sum of t. There exist a data structure of size $O(s \log(2 + t/s))$ bits, supporting the lookup of $X[i]$ and the prefix sum $\sum_{j=1}^{i} X[j]$ in $O(1)$ time, for $i = 1, .., s$. This structure requires $O(n)$ preprocessing time.*

Definition 7.2. *Dominating prefix sum is the amount of points dominated for a given point in a set, if you only consider the skyline of the points with lower y coordinates than that point in the set, added the prefix sum of the point just below the queried point, if it exist.*

Lemma 7.3. *There exist a structure given n points P where the dominating prefix sum for the point p can be queried in $O(1)$ time, with $O(n)$ bits space usage and $O(n)$ preprocessing time.*

Proof. The dominating prefix sum is a vector of non-negative integers with a total sum of n elements and with n elements. By applying Lemma 7.1 then this will result in $O(n \log(2 + n/n)) = O(n)$ bits space usage, a query time of $O(1)$ and a preprocessing time of $O(n)$. □

Lemma 7.4. *With the indexes of the points on the skyline, for a query range, with the highest and lowest y coordinate, then the skyline count can be computed by:*

$$(\textit{highSky} - \textit{lowSky}) - \textit{dps}[\textit{highSky}] - \textit{dps}[\textit{lowSky}] + 1 \text{ in } O(1) \text{ time.}$$

Proof. $(\textit{highSky} - \textit{lowSky})$ is the total amount of points that possible could be on the skyline and $\textit{dps}[\textit{highSky}] - \textit{dps}[\textit{lowSky}]$ is the amount of points dominated between the two points. By using the structure from Lemma 7.3 then the $O(1)$ query time is upheld. □

Lemma 7.5. *The structures "maximum predecessor if no equal query" (Lemma 5.3), "minimum successor if no equal query" (Lemma 5.2) and "minimum successor query" (Lemma 5.1) can all be preprocessed in $O(n)$ time and use $O(n)$ bits space, while keeping the $O(1)$ query time.*

Proof. Define a bit vector X where $X[i] = 1$ iff $sortedY[i]$ is in the first child.

A prefix sum defined as $mpne[i] = (\sum_{j=0}^{i} X[j]) - 1$ can answer "maximum predecessor if no equal query" (Lemma 5.3) for the first child, where every index without a predecessor or equal return -1 and $0 \leq i < n$.

A prefix sum defined as $msne[i] = |child| - ((\sum_{j=i}^{n-1} X[j]) - 1) - 1$ can answer "minimum successor if no equal query" (Lemma 5.2) for the first child, where every index without a successor or equal return $|child|$ and $0 \leq i < n$.

A prefix sum defined as $msq[i] = |child| - ((\sum_{j=i+1}^{n-1} X[j]) - 1) - 1$ can answer "minimum successor query" (Lemma 5.1) for the first child, where every index without a successor return $|child|$ and $0 \leq i < n$.

All these prefix sums are all n length and the total sum of them all are $n/2$ so by using Lemma 7.1 then they can all be queried in $O(1)$ time, preproccesed in $O(n)$ time and the space usage is $O(n \log(2 + (n/2)/n)) = O(n)$ bits. The same structure can be created for the second child by stating that $X[i] = 1$ iff $sortedY[i]$ is in the second child. □

Lemma 7.6. *The child to parent mapping structure mentioned in Lemma 5.4 can be preprocessed in $O(n)$ time and use $O(n)$ bits space, while keeping the $O(1)$ query time.*

Proof. Define a function $par(i)$ that for any i, $0 \leq i < |child|$, returns the index of $child.sortedY(i)$ in the $parent.sortedY$. Also Define a bit vector X of the size $|child|$ where $X[i] = par(i) - par(i-1)$ for any $0 < i < |child|$ and $X[0] = par(0)$.

A prefix sum defined as $ctp[i] = \sum_{j=0}^{i} X[j]$ can answer the child to parent mapping mentioned in Lemma 5.4, where $0 \leq i < |child|$.

This prefix sum is n length and the total sum of them all are $n * 2$ so by using Lemma 7.1 then the query time is $O(1)$, the preprocessing time is $O(n)$ and the space usage is $O(n \log(2 + (n * 2)/n)) = O(n)$ bits. □

Lemma 7.7. *There exists a structure that can perform an orthogonal range skyline count query in $O(\log n)$ time, that uses $O(n)$ words space and $O(n \log n)$ preprocessing time.*

Proof. Introducing the structure from Lemma 7.3, 7.5 and 7.6 into the previous structure from Lemma 6.2 then the query and preprocessing times are unchanged, but the space usage is reduced to $O(n)$ words. This is because every node now only uses $O(1)$ bits per point in the subtree, every layer spans $O(n)$ points and there is $O(\log n)$ layers, this gives a total of $O(n \log n)$ bits and this is $O(n)$ words. □

8 Final structure

This is the final step that implements the final structure from [Brodal and Larsen, 2014]. It replaces the binary tree structure with the Δ-tree structure from [Das et al., 2013] and [Brodal and Larsen, 2014] introduces many succinct substructures. This will result in a preprocessing time of $O(n \log n)$, space consumption of $O(n)$ words and a query time of $O(\log n / \log \log n)$.

This section is from [Brodal and Larsen, 2014]:

A helpful definition is $|\mathit{innerNode.sortedY}| = n_c$.

Definition 8.1. $\Delta = \max(2, \lceil \log^\epsilon n \rceil)$, *were* $0 < \epsilon < 1/3$.

Definition 8.2. *A Δ-tree is a balanced tree with degree Δ that stores n points in a left-to-right order with respect to the x axis.*

Definition 8.3. *The set of points contained in a subtree is called a slab. A multislab is a set of adjacent slabs from the same inner node. n_m is the total amount of points contained in the multislab.*

Definition 8.4. *All the subpoints contained in a inner node, P can be partitioned into n/Δ^2 amount of blocks, $B[]$, where $B[i] = P[(i-1)\Delta^2 + 1 \dots \min(n_c, i\Delta^2)]$ of Δ^2 size each.*

The rest of the chapter is about defining different structures and queries for each inner node, multislab and block so that all the structures at total gives $O(n)$ words space usage.

8.1 Block structure

This section contains definitions for structures that is saved globally and not anywhere in the tree. These structures will be used in the later structures as sub queries or in their preprocessing. They all achieve $O(1)$ query time and in total uses $O(n)$ bits and $O(\Delta^2 n)$ preprocessing time.

Definition 8.5. *The block signature $\sigma_v[i]$ for a block $B_v[i]$ for a inner node v is a list of pairs: For each pair p in $B_v[i]$ exists a pair (j, r) where j is the index of the child where p is stored and r is the rank of p's x-coordinate among all the other points in $B_v[i]$ stored at the same child.*

Definition 8.6. *$Below(\sigma, t, i)$ given a signature returns the number of points from $p_1 \dots p_t$ contained in slab i.*

Definition 8.7. *$Rightmost(\sigma, b, t, i, j)$ given a signature returns the index of the rightmost point from $p_b \dots p_t$ contained in the multislab $[i, j]$.*

Definition 8.8. *$Topmost(\sigma, b, t, i, j)$ given a signature returns the index of the topmost point from $p_b \dots p_t$ contained in the multislab $[i, j]$.*

Definition 8.9. *$SkyCount(\sigma, b, t, i, j)$ given a signature returns the skyline count from the subset of points $p_b \dots p_t$ contained in the multislab $[i, j]$.*

Lemma 8.10. *There exists a structure that can answer the queries from Definition 8.6, 8.7, 8.8 and 8.9 in $O(1)$ time and uses $O(n)$ bits and uses $O(\Delta^2 n)$ preprocessing time.*

Proof. The total amount of bits required for a signature is $(\Delta/2)(\log \Delta + \log \Delta^2) = O(\log^\epsilon n \log\log n)$. The size of the arguments for all the queries are at most $|\sigma| + 2\log n/2 + 2\log 2 = |\sigma| + O(\log n) = O(\log^\epsilon n \log\log n)$, so all possible combinations are $O(2^{\log^\epsilon n \log\log n})$. The space usage of the result of all the queries are $\log \Delta + 1 = O(\log\log n)$ bits big. This mean that the size of all the tables will be $O(2^{\log^\epsilon n \log\log n} \log\log n) = o(n)$ bits total.

A query in the tables takes $O(1)$ time.

The preprocessing for *Below* is a iteration over Δ^2 points in every block for each Δ amount of slabs, there are $O(2^{\log^\epsilon n \log\log n})$ different signatures so this gives $O(\Delta^2 2^{\log^\epsilon n \log\log n}) = O(\Delta^2 n)$ preprocessing time.

The preprocessing for *Rightmost* is a RMQ structure over Δ^2 points in every block for each Δ^2 amount of multislabs, by Lemma 6.1 the preprocessing of the RMQ structures can be done in $O(\Delta^2)$, there are $O(2^{\log^\epsilon n \log\log n})$ different signatures so this gives $O(\Delta^2 2^{\log^\epsilon n \log\log n}) = O(\Delta^2 n)$ preprocessing time.

The preprocessing for *Topmost* and *SkyCount* is an iteration over every Δ^4 combinations of points in every block, with a check in each of Δ^2 multislabs, there are $O(2^{\log^\epsilon n \log\log n})$ different signatures so this gives $O(\Delta^2 2^{\log^\epsilon n \log\log n}) = O(\Delta^2 n)$ preprocessing time.

This all sums up to a preprocessing time of $O(\Delta^2 n)$ for the global structure. □

8.2 Inner node structure

This section contains definitions for structures that is saved in every inner node, these structures will be used in the later structures as sub queries or in their preprocessing. They all achieve $O(1)$ query time and in total uses $O(n)$ words and $O(n \log_\Delta n)$ preprocessing time.

Lemma 8.11. *There exist a structure for every inner node that can perform $child(i)$ queries, that for every point in $child.sortedY[i]$ returns the child that contains $child.sortedY[i]$, for any $i = 0 \dots n_c - 1$. The structure have a query time of $O(1)$, space usage of $O(n_c \log \Delta)$ bits and a preprocessing time of $O(n_c)$*

Proof. The child structure can be implemented with a lookup table with $O(n_c \log \Delta)$ bits, a query time of $O(1)$ and a preprocessing of $O(n_c)$. □

This lemma is from [Raman et al., 2002]:

Lemma 8.12. *A vector $X[1 \dots s]$ of s zero-one values, with t values equal to one, can be stored in a data structure of size $O(t(1 + \log s/t)$ bits supporting rank and select queries in $O(1)$ time. A rank(i) query returns the number of ones in $X[0 \dots i]$, provided $X[i] = 1$, wheres a select(i) query returns the position of the i'th one in X. The structure have a preprocessing time of $O(s)$.*

Lemma 8.13. *There exist a structure for every inner node that can perform parent(i) query, that for any point innerNode.sortedY$[i]$ returns the index of the point in parent.sortedY, for any $i = 0 \dots n_c - 1$. The structure have a query time of $O(1)$, space usage of $O(n_c \log \Delta)$ bits and a preprocessing time of $O(n_p)$*

Proof. The amount of points in the parent subtree is n_p. In every inner child node there is created a bit-vector X with size n_p, where if $X[i] = 1$ then *parent.sortedY*$[i]$ exists in *child.sortedY*. By doing the select query from Lemma 8.12 then the index of the parent node will be returned, because if $0 \leq j \leq k < n_c$ then *parent*$(j) \leq$ *parent*(k). The lemma states that the structure is capable of querying in $O(1)$ time, and a preprocessing time of $O(n_p)$. The space usage is $O(t(1 + \log s/t) = O(n_c(1 + \log n_p/n_c) = O(n_c \log \Delta)$ bits, because there is a Δ factor in difference between n_c and n_p at every layer. □

Lemma 8.14. *Every inner node have an array of signatures for its blocks $\sigma(1 \dots \lceil n/\Delta^2 \rceil)$, the space usage is $O((n_c/\Delta^2)\Delta^2 \log \Delta) = O(n_c \log \Delta)$ bits.*

Lemma 8.15. *There exist a structure for every inner node that can perform predecessor(i, j)/successor(i, j) queries, that for any point sortedY$[j]$ returns the predecessor/successor of sortedY$[j]$ in the child i'th sorted y list, for any $j = 0 \dots n_c - 1$ and $i = 0 \dots \Delta - 1$. The structure have a query time of $O(1)$, space usage of $O(n_c)$ bits and a preprocessing time of $O(n_c)$.*

Proof. By defining an array X of size $\Delta\lceil n_c/\Delta^2 \rceil$, so for every $X[k][i]$ is the number of points in block k that exists in the i'th child, then the prefix sum, *prefixSum*$(k, i) = \sum_{r=0}^{k} X[r][i]$, of this structure can be stored as described in Lemma 7.1 using $O(\Delta(s \log(2 + t/s))) = O(\Delta((n_c/\Delta^2) \log 2 + n_c/(n_c/\Delta^2))) = O(\Delta((n_c/\Delta^2) \log \Delta^2)) = O(n_c)$ bits space, a query time of $O(1)$ and a preprocessing time of $O(n_c)$. Answering the query is then *prefixSum*$(\lceil j/\Delta^2 \rceil - 1, i)$ + *Below*$(\sigma(\lceil j/\Delta^2 \rceil), 1 + (j - 1 \bmod \Delta^2), i)$, that can be answered in $O(1)$ time and were the *Below* structure is already counted into the global tables. □

Lemma 8.16. *There exist a structure for every inner node that can perform rightmost(i, j) query, that for any interval sortedY$[i \dots j]$ returns the index of the point with the maximum x-value in the interval, for any $0 \leq i \leq j < n_c$. The structure have a query time of $O(1)$, space usage of $O(n_c)$ bits and a preprocessing time of $O(n_c)$.*

Proof. The RMQ structure from Lemma 6.1 can be used on the x-coordinates of *sortedY* to achieve $O(1)$ query time, space usage of $O(n_c)$ bits and a preprocessing time of $O(n_c)$. □

Lemma 8.17. *There exist a structure for every inner node that can perform a skyCount(i, j) query structure, that for any interval sortedY$[i \dots j]$ returns the skyline count if only the points in the interval were considered, for any $0 \le i \le j < n$. The structure have a query time of $O(1)$, space usage of $O(n_c)$ bits and a preprocessing time of $O(n_c)$.*

Proof. By using the dominating prefix sum structure from Lemma 7.4 then the $O(1)$ query time , space usage of $O(n_c)$ bits and a preprocessing time of $O(n_c)$ is achieved. □

Lemma 8.18. *All the structures mentioned in Lemma 8.11, 8.13, 8.14, 8.15, 8.16 and 8.21, can all be queried in $O(1)$ time. There is a total space use of $O(n)$ words and a preprocessing time of $O(n \log_\Delta n)$ in the whole tree.*

Proof. The query time of all the structures are proved in each Lemma individually. The total size of all the structures in each inner node is $O(n_c \log \Delta)$ bits and every layer in the Δ-tree spans n elements, so every layer uses $O(n \log \Delta)$ bits. There are $O(\log_\Delta(n))$ layers so the total size of all the inner node structures are $O(n \log \Delta \log_\Delta(n)) = O(n \log n)$ bits that is $O(n)$ words. If we let the child to parent mapping of Lemma 8.13 count in the parent node, because there will only be a child to parent mapping if there is a parent, then for every structure the preprocessing time of $O(n_c)$, every layer spans $O(n)$ points and there are $O(\log_\Delta(n))$ layers this will total up to $O(n \log_\Delta n)$ preprocessing time. □

8.3 Multislab structure

This section contains definitions for structures that is saved in every inner node one for each Δ^2 multislabs in that inner node. This is the last structures that need to be defined before the orthogonal range skyline counting query can be implemented. They all achieve $O(1)$ query time and in total uses $O(n)$ words and $O(nlog_\Delta(n))$ preprocessing time.

Lemma 8.19. *There exist a structure for every inner node that can perform rightmost(i, j, b, t) query structure, that for any interval $B[i \dots j]$ returns the point with the maximum x-value in the interval that is within the multislab $[b, t]$, for any $0 \le i \le j < n/\Delta^2$ and $0 \le b \le t < \Delta$. If no point exist then -1 is returned. The structure have a query time of $O(1)$, space usage of $O(n_c/\Delta^2)$ bits and a preprocessing time of $O(n_c/\Delta^2)$.*

Proof. By defining an array X of size n/Δ^2 for every multislab, where $X[s]$ returns the maximum point with respect to x axis out of all the points in block $B[s]$ that is within multislab $[b, t]$, then Lemma 6.1 can be used to create a RMQ over all those points and returns the index of the block, with a query time of $O(1)$, space usage of $O(n_c/\Delta^2)$ and a preprocessing time of $O(n_c/\Delta^2)$.

The query then first queries the RMQ structure for the block l that contains the point we search for and then queries the block to get the index of the point. $(l-1)\Delta^2$ gives the count of all the points below block l in *sortedY* within the multislab $[b, t]$ and a query in the global table *Rightmost*$(\Sigma[l], 1, \Delta^2, b, t)$ (Lemma 8.7) will return the index of the rightmost point of all the points in block l if only the points in multislab $[b, t]$ were considered, so combined they give the answer to the query. □

Lemma 8.20. *There exist a structure for every inner node that can perform topmost*(i, j, b, t) *query structure, that for any interval* $B[i \dots j]$ *returns the point with the maximum y-value in the interval that is within the multislab* $[b, t]$*, for any* $0 \leq i \leq j < n/\Delta^2$ *and* $0 \leq b \leq t < \Delta$*. If no point exist then* -1 *is returned. The structure have a query time of* $O(1)$*, space usage of* $O(n_c/\Delta^2)$ *bits and a preprocessing time of* $O(n_c/\Delta^2)$*.*

Proof. By defining an array X of size n/Δ^2 for every multislab, where $X[s] = s$ if there exist a point in $B[s]$ that is within multislab $[b, t]$, then Lemma 6.1 can be used to create a RMQ over all those points and returns the index of the block, with a query time of $O(1)$, space usage of $O(n_c/\Delta^2)$ and a preprocessing time of $O(n_c/\Delta^2)$.

The query then first queries the RMQ structure for the block l that must be the block with the point with the highest index in *sortedY* and by that the topmost point, within the multislab $[b, t]$. $(l-1)\Delta^2$ gives the count of all the points below block l in *sortedY* and a query in the global table *Topmost*$(\Sigma[l], 1, \Delta^2, b, t)$ (Lemma 8.8) will return the index of the topmost point of all the points in block l if only the points in multislab $[b, t]$ were considered, so combined they give the answer to the query. □

Lemma 8.21. *There exist a structure for every inner node that can perform a skyCount*(i, j, b, t) *query structure, that for any interval* $B[i \dots j]$ *returns the skyline count if only the points in the interval that is within the multislab* $[b, t]$ *were considered, for any* $0 \leq i \leq j < /\Delta^2$ *and* $0 \leq b \leq t < \Delta$*. The structure have a query time of* $O(1)$*, space usage of* $O((n_c/\Delta^2) \log \Delta^2)$ *bits and a preprocessing time of* $O(n/\Delta^2)$*.*

Proof. By defining an array X of size n/Δ^2 for every multislab, where $X[s] = \mathit{skyCount}(\sigma(s), 1, \Delta^2, b, t)$ (From Lemma 8.9), then Lemma 7.1 can be used to create a Prefix Sum structure, with a query time of $O(1)$, space usage of $O(s \log(2 + t/s)) = O((n_c/\Delta^2) \log(2 + n_c/(n_c/\Delta^2))) = O((n_c/\Delta^2) \log \Delta^2)$ and a preprocessing time of $O(n/\Delta^2)$, for one structure for one multislab. Because t sums up to n_c, because there can maximum be Δ^2 points at each index and there is n/Δ^2 indexes.

By defining another array Y of size n/Δ^2 for every multislab, where $Y[s] = y$ where y is the amount of points dominated by $\mathit{Skyline}(\sigma(s), 1, \Delta^2, t, b)$ from $\mathit{Skyline}(\sigma(1 \ldots s-1), 1, \Delta^2, t, b)$, then Lemmas 7.1 and 7.3 can be used to create a Prefix Sum structure, with a query time of $O(1)$, space usage of $O(s \log(2 + t/s)) = O((n_c/\Delta^2) \log(2 + n_c/(n_c/\Delta^2))) = O((n_c/\Delta^2) \log \Delta^2)$ and a preprocessing time of $O(n/\Delta^2)$, for one structure for one multislab. Because t sums up to n_c, because every point can only be dominated once.

The query will then be answered by a $X[s] + (\mathit{prefixSumX}(t) - \mathit{prefixSumX}(k) - (\mathit{prefixSumY}(t) - \mathit{prefixSumY}(k+1)))$ where $k = \lceil \mathit{rightmost}(i, j, b, t) \rceil$ (Form Lemma 8.19), in $O(1)$ time. The space usage of the structure for one multislab is $O((n_c/\Delta^2) \log \Delta^2)$ bits and preprocessing is $O(n/\Delta^2)$. □

Lemma 8.22. *All the structures mentioned in Lemma 8.19, 8.20 and 8.21, can all be queried in $O(1)$ time. The total space usage for the whole tree structure for all the multislab structures are $O(n)$ words and have a preprocessing time of $O(nlog_\Delta(n))$*

Proof. There exist Δ^2 multislabs in every node, so the space usage for every inner node is $O(\Delta^2(n_c/\Delta^2) \log \Delta^2) = O(n_c \log \Delta)$ bits, at every layer of the tree spans $O(n)$ nodes, so the space usage for every layers is $O(n \log \Delta)$. There are $O(log_\Delta(n))$ layers so the total size of all the multislab structures are $O(n \log \Delta log_\Delta(n)) = O(n \log n)$ bits and this is $O(n)$ words.

There exist Δ^2 multislabs in one inner node, so the preprocessing time for all the multislabs in one inner node is $O(n_c)$. One layer in the tree spans n nodes and there are $O(log_\Delta(n))$ layers, so the total preprocessing time of all the multislab structures is $O(nlog_\Delta(n))$. □

8.4 Orthogonal range skyline counting query

Lemma 8.23. *There exists a structure that can perform an orthogonal range skyline count query in $O(\log n)$ time, that uses $O(n)$ words space and has a preprocessing time of $O(n \log n)$*

Proof. Only the differences compared to the previous structures will be mentioned here.

When the super query iterates through the inner nodes of the high and low x paths, then there will be at most one multislab there is completely contained in the query range and at most one slab that is partially contained. The partial contained slab is the next or previous iteration of the query and will be handled there, so the subquery is on the maximal multislab contained in the query range for an inner node.

As in the previous steps the iteration of the paths will be done from left to right, so there can be kept track of $lowY$ through the query. So every subquery is performed on a inner node in the paths, with a $highY$, $lowY$ and a slab interval $[i, j]$. The query will be done in a series of steps that all takes $O(1)$ time:

First if the subquery only query one block $B[\lceil highY/\Delta^2 \rceil]$ then that block only needs to be queried: $\mathit{SkyCount}(\Sigma(\lceil highY/\Delta^2 \rceil), 1 + (lowY - 1 \bmod \Delta^2), 1 + (highY - 1 \bmod \Delta^2), i, j)$ (From Lemma 8.9). Else if there is more than one block then the sub query will go through these steps:

1. The skyline count is calculated for the top block in the query range by: $\mathit{SkyCountTop} = \mathit{Skycount}(\sigma(\lceil highY/\Delta^2 \rceil), 1, 1 + (highY - 1 \bmod \Delta^2), i, j)$ (From Lemma 8.9).

2. The index of the rightmost point in the top block is calculated by: $p_1 = \mathit{Rightmost}(\sigma(\lceil highY/\Delta^2 \rceil), 1, 1 + (highY - 1 \bmod \Delta^2), i, j) + \Delta^2 \lceil highY/\Delta^2 \rceil$ (From Lemma 8.7).

3. The slab containing p_1 is calculated by: $k_1 = \mathit{child}(p_1)$ (From Lemma 8.11).

4. The skyline count in the slabs $[k+1, j]$ and all the blocks excluding top and bottom block $b[\lceil lowY/\Delta^2 \rceil + 1 \ldots \lceil highY/\Delta^2 \rceil - 1]$ is calculated by: $\mathit{SkylineCountMiddle} = \mathit{SkyCount}(\lceil lowY/\Delta^2 \rceil + 1, \lceil highY/\Delta^2 \rceil - 1, k + 1, j)$ (From Lemma 8.21).

5. The index of the topmost point in the slabs $[k+1, j]$ and all the blocks excluding top and bottom block $b[\lceil lowY/\Delta^2 \rceil + 1 \ldots \lceil highY/\Delta^2 \rceil - 1]$ is calculated by: $p_2 = \mathit{Topmost}(\lceil lowY/\Delta^2 \rceil + 1, \lceil highY/\Delta^2 \rceil - 1, k+1, j)$ (From Lemma 8.20).

6. The index of the rightmost point in the slabs $[k+1, j]$ and all the blocks excluding top and bottom block $b[\lceil lowY/\Delta^2 \rceil + 1 \ldots \lceil highY/\Delta^2 \rceil - 1]$ is calculated by: $p_3 = \mathit{Rightmost}(\lceil lowY/\Delta^2 \rceil + 1, \lceil highY/\Delta^2 \rceil - 1, k + 1, j)$ (From Lemma 8.19).

7. The slab containing p_3 is calculated by: $k_3 = \mathit{child}(p_3)$ (From Lemma 8.11).

8. The skyline count is calculated for the bottom block in the multislab $[k_3{+}1, j]$ in the query range by: $\mathit{SkyCountBottom} = \mathit{Skycount}(\sigma(\lceil \mathit{lowY}/\Delta^2\rceil), 1 + (\mathit{lowY} - 1 \bmod \Delta^2), \Delta^2, k_3 + 1, j)$ (From Lemma 8.9).

9. The index of the topmost point in the bottom block is calculated by: $p_4 = \mathit{Topmost}(\sigma(\lceil \mathit{lowY}/\Delta^2\rceil), 1 + (\mathit{lowY} - 1 \bmod \Delta^2), \Delta^2, k_3 + 1, j) + \Delta^2\lceil \mathit{lowY}/\Delta^2\rceil$ (From Lemma 8.8).

10. The skyline count is calculated for the points bellow p_1 and above p_2 in slab k_1 in the query range by:

    $\mathit{SkyCountK}_1 = \mathit{Skycount}(\mathit{successor}(p_2, k_1), \mathit{predecessor}(p_1, k_1)){-}1$ (From Lemma 8.21 and 8.15), the -1 is because p_1 should not be counted twice.

11. The skyline count is calculated for the points bellow p_3 and above p_4 in slab k_1 in the query range by:

    $\mathit{SkyCountK}_3 = \mathit{Skycount}(\mathit{successor}(p_4, k_3), \mathit{predecessor}(p_3, k_3)){-}1$ (From Lemma 8.21 and 8.15), the -1 is because p_3 should not be counted twice.

12. The skyline count for the slabs $[i, j]$ between highY and lowY is calculated by: $\mathit{SkylineCount} = \mathit{SkyCountTop} + \mathit{SkylineCountMiddle} + \mathit{SkyCountBottom} + \mathit{SkyCountK}_1 + \mathit{SkyCountK}_3$.

There are some special cases. If p_1 does note exist then $k_1 = i - 1$ and $\mathit{SkyCountK}_1$ is not calculated. If P_4 does not exist then $p_4 = \mathit{lowY}$ when calculating $\mathit{SkyCountK}_3$. If p_2 and p_3 does not exist then $\mathit{SkyCountK}_3$ and $\mathit{SkylineCountMiddle}$ are not computed, the leftmost slab of $\mathit{SkyCountBottom}$ is the $k_1 + 1$ and when calculating $\mathit{SkyCountK}_1$ then $p_2 = p_4 + 1$.

All these steps take $O(1)$ time each, and because the height of the tree is $O(n \log n/ \log\log n)$ and the query is bounded by the high and low x paths then the total time of the orthogonal range skyline counting query is $O(n \log n/ \log\log n)$.

The total space use is the sum of the global tables $O(n)$ bits (Lemma 8.10), the inner node structure $O(n)$ words (Lemma 8.18) and the multislab structures $O(n)$ words (Lemma 8.22), that totals up to a $O(n)$ words space usage for the final structure.

The preprocessing time of all the substructures are all $O(n \log_\Delta n)$ (Lemma 8.10, 8.18 and 8.22) but this is less than the initial sorting of the input set of points, so the preprocessing time of the final structure is $O(n \log n)$.

$\square$

9 Tests

9.1 Test machine

OS	Windows 7 64 bit
CPU	Intel(R) Core(TM) 2 Quad CPU Q9650 3.00GHz
RAM	4 GB
L1 D-cache	4 * 32 KB
L1 I-cache	4 * 32 KB
L2 cache	2 * 6144 KB
L3 cache	Not enabled

9.2 Measurement setup

During all tests the space usage, query time, preprocessing time and number of activated critical sections are measured. Where critical sections are the time consuming sections of the query. So RMQ simples and Fractionals critical sections is when a sub query is activated and the Naive has a critical section for every iteration. The critical sections is usually not shown in this test section but is used to verify that the tests run the same input, because the critical sections of RMQ simple and Fractional will be the same on the same input.

The critical sections are counted in a local variable during the query and then accessed after the query, at the beginning of a query it gets reset. These very simple operations happens during the query time measurements but should not have any visible effect on the measurements.

Query time and preprocessing time is a windows operation system specific "clock t" system clock that measures "Clock ticks", the length of "Clock ticks" are system dependent, but because the same machine were used for every test then "Clock ticks" are constant. In every test the measurements only measures around the preprocessing and query calls, where the same query usually gets called many times and the measurement is measuring the running time of all the queries together. This is because the query time is usually so low that multiple consecutive calls need to be made, to have some good data to analyze. Consecutive query calls will be influenced by caching, so even though this will influence the algorithms getting tested, then the difference in effect is not expected to be major.

The size of a structure is measured with a recursive call, in the case of RMQ simple and Fractional through their tree structures, where every node sums up its total size, and in the Naive case it is the size of the sorted list. There is a risk that some parts of the structure were forgotten, but there were no feature that could calculate the size of an object, its internal substructures and every other object it were dependent on.

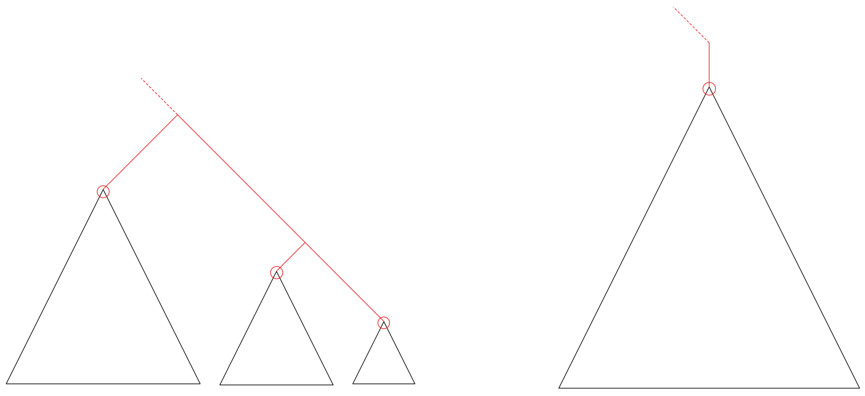

Figure 7: Two different cases of a right part of a tree, could be RMQ simple or Fractional. The red circles shows possible critical sections for both trees. Notice that the only difference between the two cases could be one point in the query range, so that the three subtrees where queried as on big tree. This means that even though two query ranges are quite close it could mean a big difference in querying time.

9.3 Test data

The input for all the algorithms are a set of points, not necessarily sorted, to be able to generate the structures, for preprocessing, and the two points that represent the orthogonal range, for the orthogonal range skyline counting query. This sub section describes, through pseudo code, how all the input set of points is generated.

The "generateWorstCase" pseudo code will generate a set of points where every point, in the query range, is on the skyline. If the whole point set is queried, then it will result in the maximum amount of activated critical sections for that input size. If the query is smaller then, in the case of RMQ simple and Fractional as shown in Figure 7, a small change to the query range could mean a big change in the amount of activated critical sections.

Algorithm 10 generateWorstCase(int *size*)

```
$worstCaseSet \leftarrow point[size]$
**for** $i \leftarrow 0$ **to** $size - 1$ **do**
    $worstCaseSet[i] \leftarrow point(i, size - i)$
**return** $worstCaseSet$
```

The "generateBestCase" pseudo code will generate a set of points where only one point is on the skyline, as long as at least one point is in the query range. This results in the least amount of critical sections getting activated as possible, no matter the query region.

Algorithm 11 generateBestCase(int *size*)

```
$bestCaseSet \leftarrow point[size]$
for $i \leftarrow 0$ to $size - 1$ do
    $bestCaseSet[i] \leftarrow point(i, i)$
return $bestCaseSet$
```

The "generateBestToWorstCase" pseudo code will generate a set of points, where there at a maximum query region will be *skylineSize* points on the skyline. Because the best and worst case is dependent on how many critical section gets accessed during the query, then with this code there is a control over how time consuming the point set could be. But keep in mind the tree structure of RMQ simple and Fractional, where many new points to the skyline could be added to a subtree that is already activated and by that not adding to the query time.

Algorithm 12 generateBestToWorstCase(int *size*, int *skylineSize*)

```
$bestToWorstCaseSet \leftarrow point[size]$
for $i \leftarrow 0$ to $size - 1$ do
    if $i < skylineSize$ then
        $bestToWorstCaseSet[i] \leftarrow point(size - i, i)$
    else
        $bestToWorstCaseSet[i] \leftarrow point(size - i, size - i)$
return $bestToWorstCaseSet$
```

The "generateLogarithmicSizes" pseudo code does not generate a set of points but it generate a list of sizes. It is given a max size, of a tree, and returns a list of sizes where every size will activate one more subtree in RMQ simple and Fractional queries. So the difference in size between the first half of the points will be increasing binary like $1, 2, 4, 8, 16$ because the distance between them is $0, 1, 2, 4, 8$. This is what the first for loop does.

The distance between the last half of the points will be decreasing binary starting from the value of the previous distance like $1, 2, 4, 8, 16, 24, 28, 30, 31$ because the distances is now $0, 1, 2, 4, 8, 8, 4, 2, 1$.

The last for loop will put a point in between all the existing points that have the average value of the point on its side, like:
$1, 1, 2, 3, 4, 6, 8, 12, 16, 20, 24, 26, 28, 29, 30, 30, 31$.

These sizes will usually be used as in "exampleOfUse" pseudo code. If the sizes only contained the sizes added in the first two for loops, then the points of iteration i will be able to trigger one more subtree in RMQ simple and Fractional, on a query over all the points, than the points from iteration $i-1$. The last for loop is there to add some sizes in between so the tests are able to verify that this is the case.

Algorithm 13 generateLogarithmicSizes(int *maxSize*)

```
$sizesBeta \leftarrow List$
for $i \leftarrow 1; i < maxSize/2; i \leftarrow i * 2$ do
    $sizesBeta.pushBack(i)$
$currentSize \leftarrow |sizesBeta|$
$newValue \leftarrow sizesBeta[|sizesBeta| - 1]$
for $i \leftarrow 1$ to $currentSize$ do
    $expandingValue \leftarrow sizesBeta[currentSize - i]$
    $newValue \leftarrow newValue + expandingValue$
    $sizesBeta.pushBack(newValue)$
$sizes \leftarrow List$
for $i \leftarrow 0$ to $|sizesBeta| - 1$ do
    $sizes.pushBack(sizesBeta[i])$
    $sizes.pushBack(sizesBeta[i] + (sizesBeta[i+1] - sizesBeta[i])/2)$
return $sizes$
```

Algorithm 14 exampleOfUse(int *size*)

```
$sizes \leftarrow generateLogarithmicSizes(size)$
for $i \leftarrow 0$ to $|sizes| - 1$ do
    $points \leftarrow generateBestToWorstCase(size, sizes[i])$
    test RMQ Simple or Fractional
```

9.4 Worst case

The purpose of this test is to observe the effect of an increased input size in the worst case scenario. The input point set is generated using the "generateWorstCase" pseudo code, where every point in the set is on the skyline if they are inside the query range. The input for the orthogonal range skyline counting query range is two points that spans the whole input point set, the max query. There will be an individual test for each algorithm. The test performs 66 iterations each with a bigger size, starting from 500 and increasing with 500 for each step and ending at 33000 points. At every iteration there will be generated a worst case point set, a structure of the given algorithm and performed 100000 identical max queries.

It is expected that the Naive algorithms preprocessing step will be faster than the two other algorithms, not only because the theoretical preprocessing time is $O(n \log n)$ and the others are $O(n \log^2 n)$, but because there is far smaller constant hidden in the preprocessing time. Where the RMQ simple probably will have a smaller preprocessing time than Fractional, because Fractional have more structures that needs to be preprocessed.

The size is also expected to be in the same order. Because Naive have much fewer structures than RMQ simple, that have fewer structures than Fractional.

The query time of Naive is expected to be much higher than RMQ simple, that will be higher than Fractional. Because in the worst case the maximum amount of critical sections gets accessed, and Fractional and RMQ simple have way less than Naive, and Fractionals critical sections takes less time than RMQ simples does.

Figure 8 shows the query time of all the algorithms in one graph. Here we can conclude that the Naive quickly becomes slower than the two other algorithms. Notice that all the graphs have a logarithmic x-axis.

Figure 9 shows just the RMQ simple and the Fractional cascading algorithms query time, and we can conclude that in the worst case Fractional is faster than RMQ simple. Because the major difference between them is the binary searches in every subtree in the RMQ simple query, then when we query every subtree we will get the biggest difference between the two algorithms. Later there will be a test in how few subtrees need to be queried before the RMQ simple is better than Fractional.

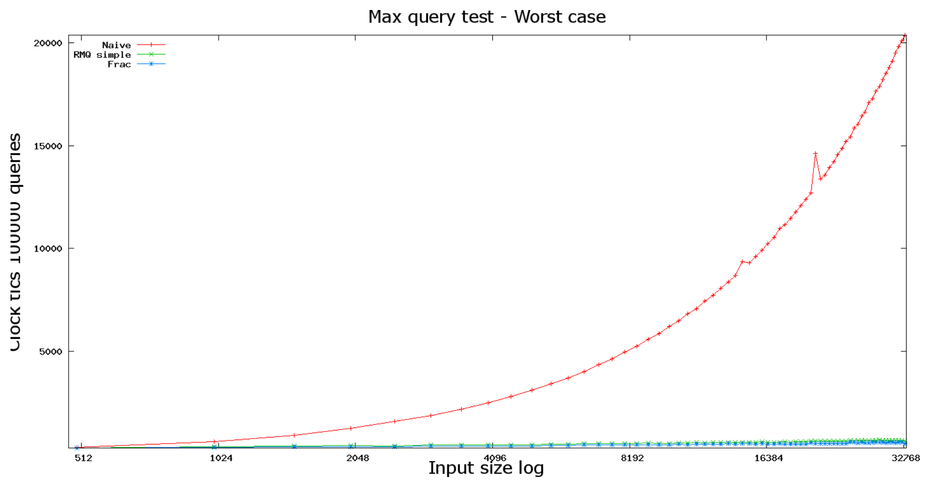

Figure 8: Query time of all the algorithms, worst case.

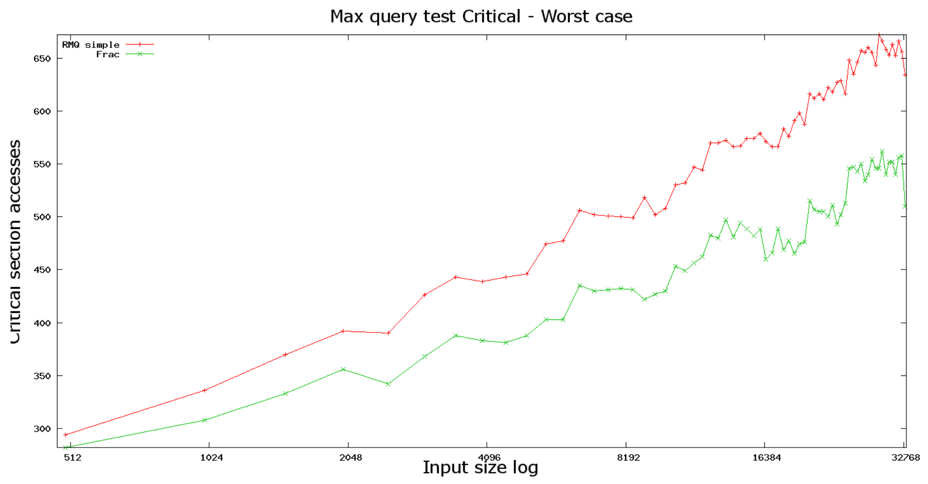

Figure 9: Query time of RMQ simple and Fractional the algorithms, worst case.

Figure 10 shows the Fractional query time divided by its theoretical running time $\log n$ and Figure 11 shows fractional divided by $\log^2 n$. It does seem that the running time is closer to $\log n$ than $\log^2 n$ and this holds with the theory.

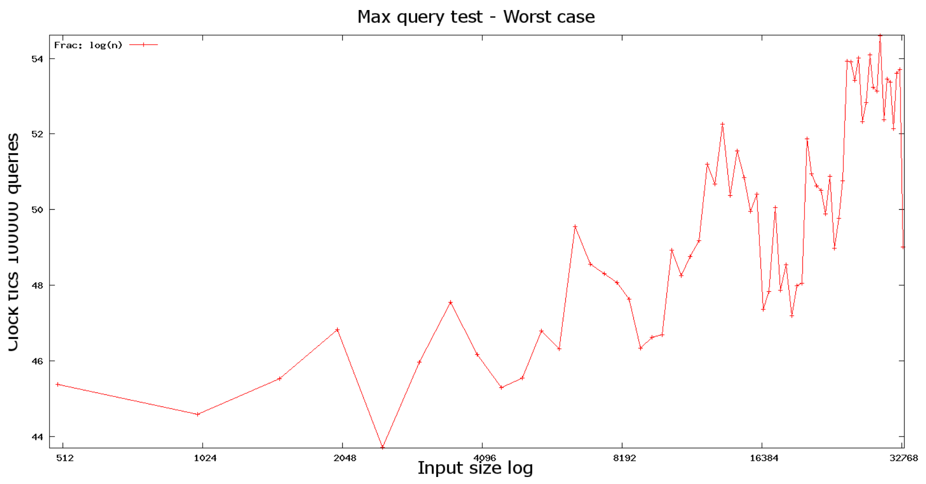

Figure 10: Fractional divided by its theoretical runningtime $\log n$, worst case.

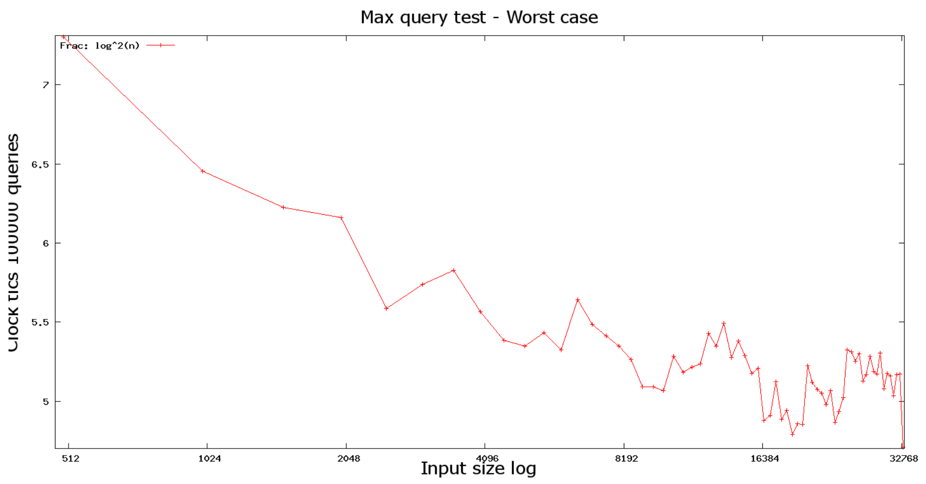

Figure 11: Fractional divided by $\log^2 n$, worst case.

Figure 12 shows the RMQ simpel query time divided by the theoretical running time $\log^2 n$ and Figure 13 shows the RMQ simpel query time divided by $\log n$. It shows that the running time is somewhere between $\log^2 n$ and $\log n$, and it looks surprisingly similar in development to the Fractional. But figure 14 where RMQ simple query time is subtracted by Fractional query time, notice the logarithmic x axis, shows that there is a logarithmic difference between the two algorithms worst case running times and that holds with the theory.

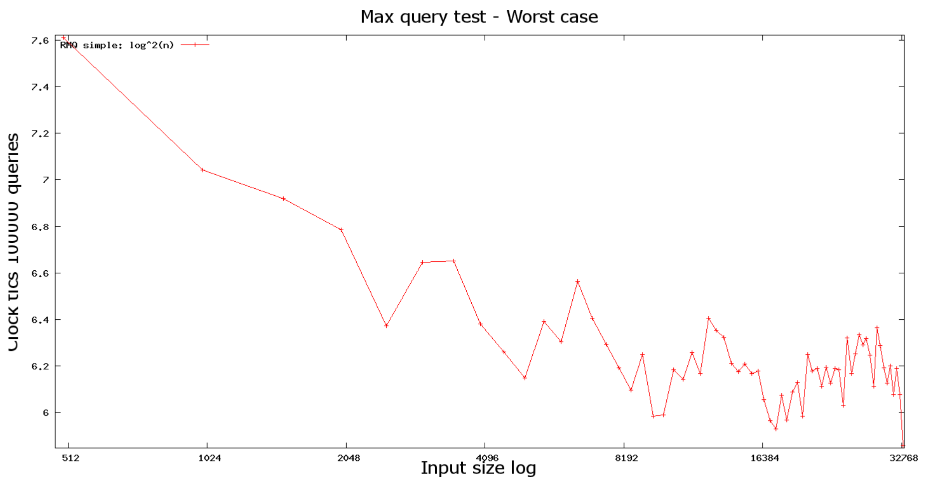

Figure 12: RMQ simple divided by its theoretical runningtime $\log^2 n$, worst case.

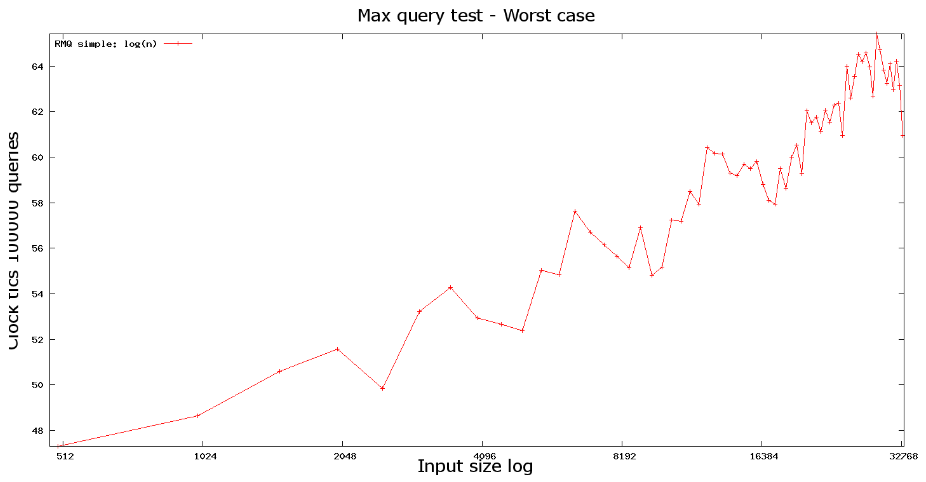

Figure 13: RMQ simple divided by $\log n$, worst case.

Figure 15 shows the Naive query time divided by the theoretical running time n, beside a couple of non reproducible anomalies then it follows the theoretical running time.

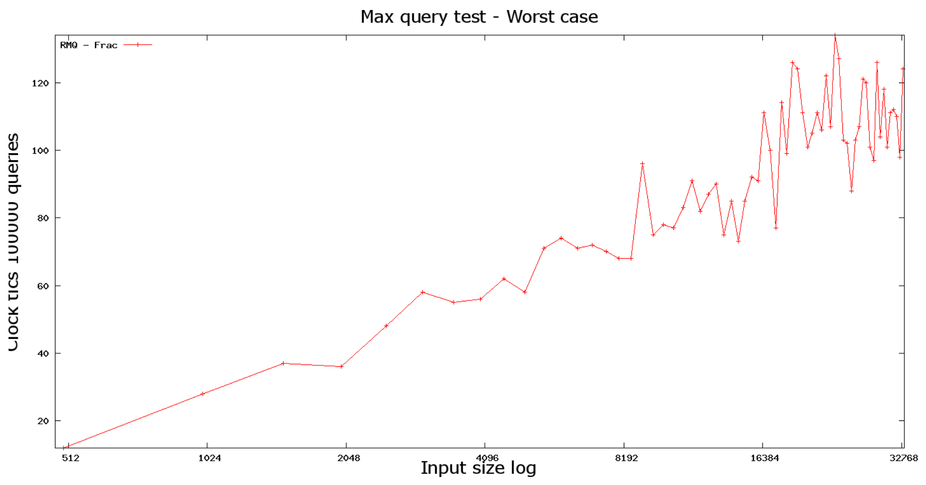

Figure 14: RMQ simple query time subtracted by Fractional query time, worst case.

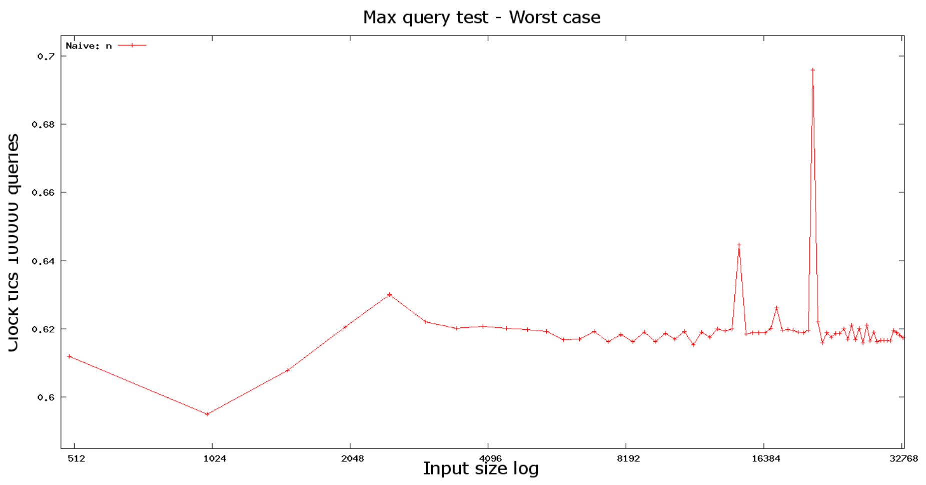

Figure 15: Naive divided by its theoretical runningtime n, worst case.

Figure 16 shows the structure sizes of all the algorithms and here we can see that Fractional is bigger than RMQ simple, that is bigger than Naive, as expected. Figure 17 shows the structure sizes of all the algorithms divided by their theoretical sizes. We can conclude that the Naive algorithm fits quite well with the theoretical bound. While both RMQ simpel and Fractional seems to be a bit lower than there theoretical size. Figure 18 shows Fractional and RMQ simpel both divided by $\log(n)\log(\log(n))n$, and it seems to match quite good. There were not found an answer to why this is the case and it is better than the theory.

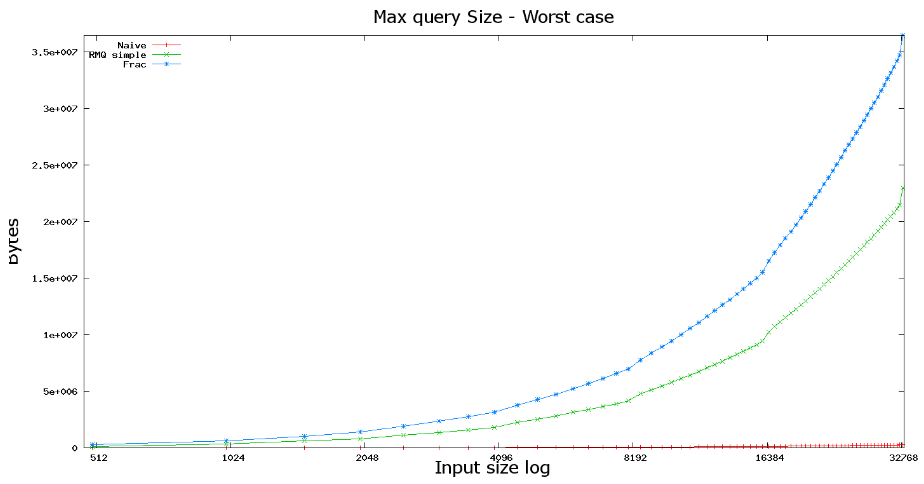

Figure 16: Structure size of all algorithms, worst case.

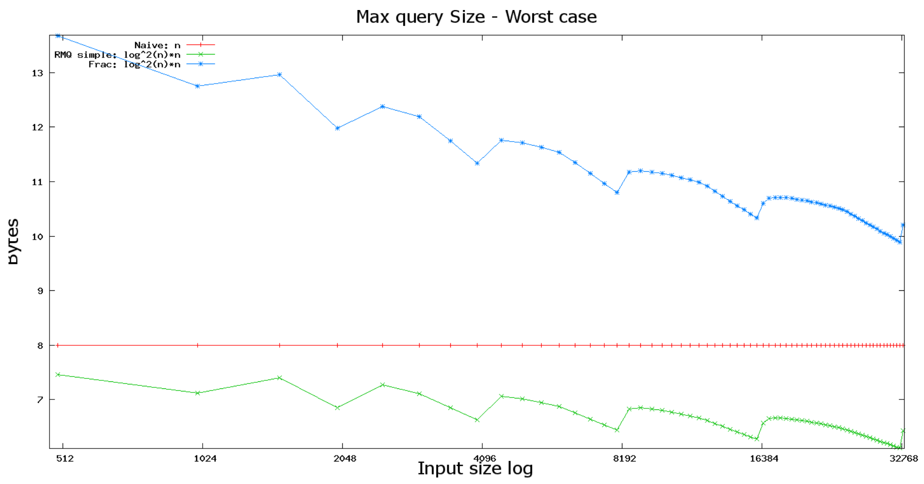

Figure 17: Structure size of all algorithms divided by their theoretical size, worst case.

Figure 19 shows the preprocessing time of all the algorithms and it shows as expected that Naive is faster than RMQ simple that is faster than Fractional. Figure 20 shows the preprocessing running time of RMQ simple and Fractional by their theoretical running times, $n \log^2 n$, it shows that they are both faster than the theoretical running times. Figure 21 shows the preprocessing running time of RMQ simple and Fractional divided by, $n \log n$, and this seems to fit better. There were not found an answer to why this is the case and it is faster than the theory.

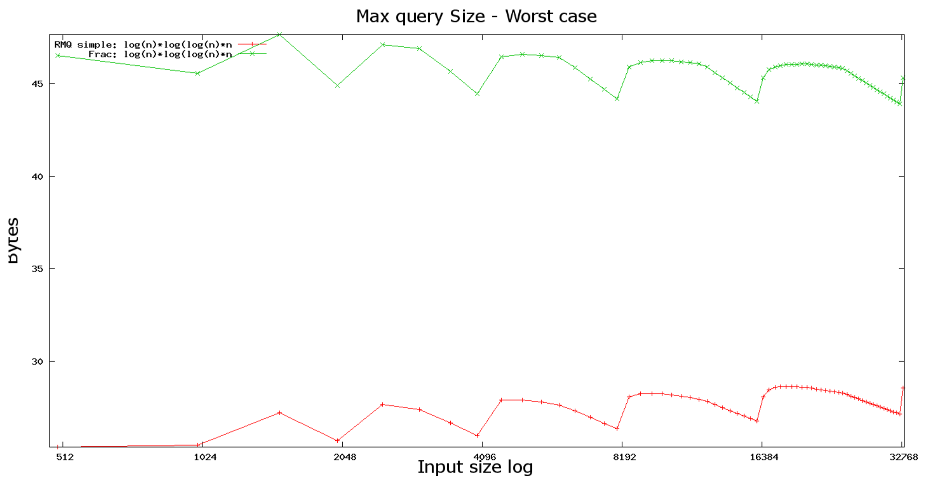

Figure 18: RMQ simple and Fractional divided by $(\log n log \log n)n$, worst case.

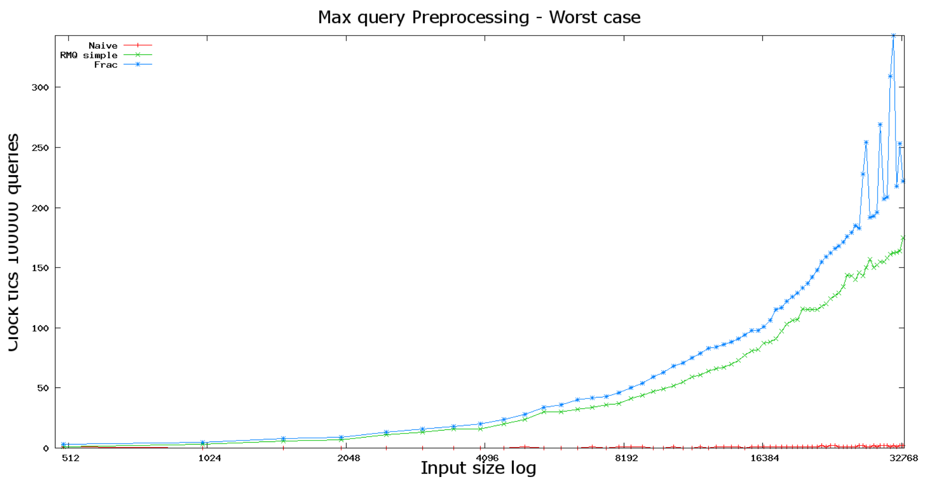

Figure 19: Preprocessing running time of all algorithms, worst case.

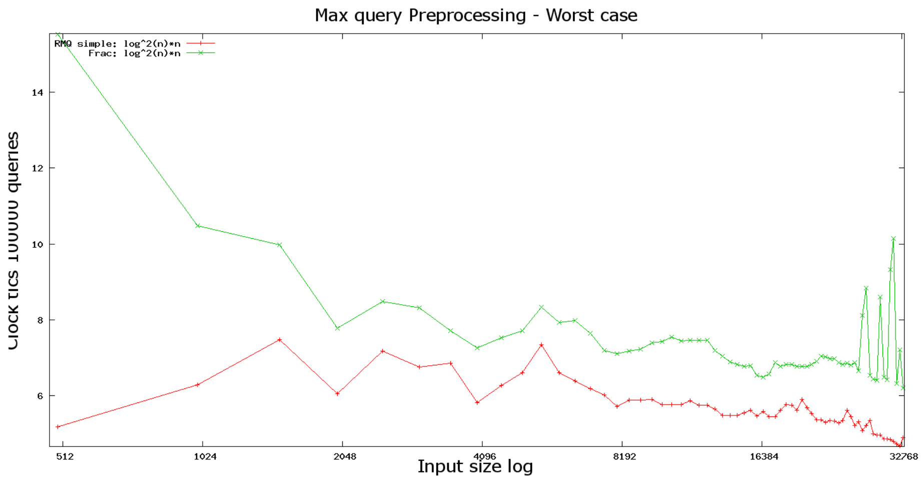

Figure 20: Preprocessing running time of RMQ simple and Fractional by their theoretical running times $n \log^2 n$, worst case.

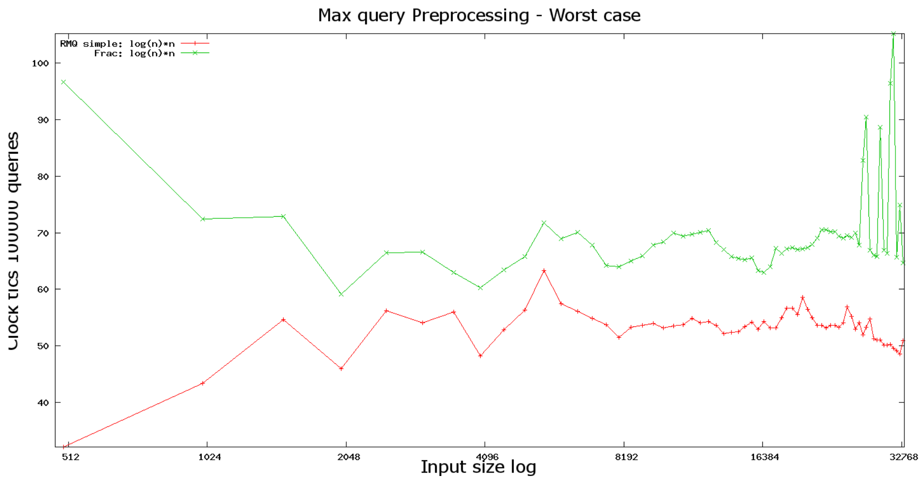

Figure 21: Preprocessing running time of RMQ simple and Fractional divided by $n \log n$, worst case.

9.5 Best case

The purpose of this test is to observe the effect of an increased input size in the best case scenario. The input point set is generated using the "generateBestCase" pseudo code, where there will only be one point on the skyline as long as there is at least one point in the query range. The input for the orthogonal range skyline counting query range is two points that spans the whole input point set, the max query. There will be an individual test for each algorithm. The test performs 66 iterations each with a bigger size, starting from 500 and increasing with 500 for each step and ending at 33000 points. At every iteration there will be generated a worst case point set, a structure of the given algorithm and performed 100000 identical max queries.

Only the parts of the results from this test that deviates from the previous test will be shown. If the lack of deviation is surprising then it will be commented.

It is expected that the preprocessing time and size will be unchanged, because they do not depend on the values of the points added. While the query time is expected to be faster in every case and the improvement will be biggest with RMQ simple. This is because in the best case very few critical sections will be triggered and these are the most time consuming parts of the query in all the algorithms. Naives critical section is every iteration, so it will only be limited by the points in the query range. RMQ simples and Fractionals critical sections are when a sub query is triggered, this only happens if the sub query will add to the skyline and this can only happen once. So because there will only be triggered one critical section then the query time will fall for RMQ simple and Fractional. It is also excpected that RMQ simple will be faster than Fractional, because Fractional have more complicated main query that results in a higher constant than RMQ simple.

Figure 22 shows all the query times of the algorithms in the best case, if it is compared to Figure 8 from worst case, then it can be seen that at least Naive have become faster, and that apparently the code adding skyline points had an effect on the running time. Figure 23 shows the query time of Fractional and RMQ simpel. RMQ simpel have a better running time than Fractional, as expected. If it is compared to Figure 9 then there is time gain on the queries, also as expected.

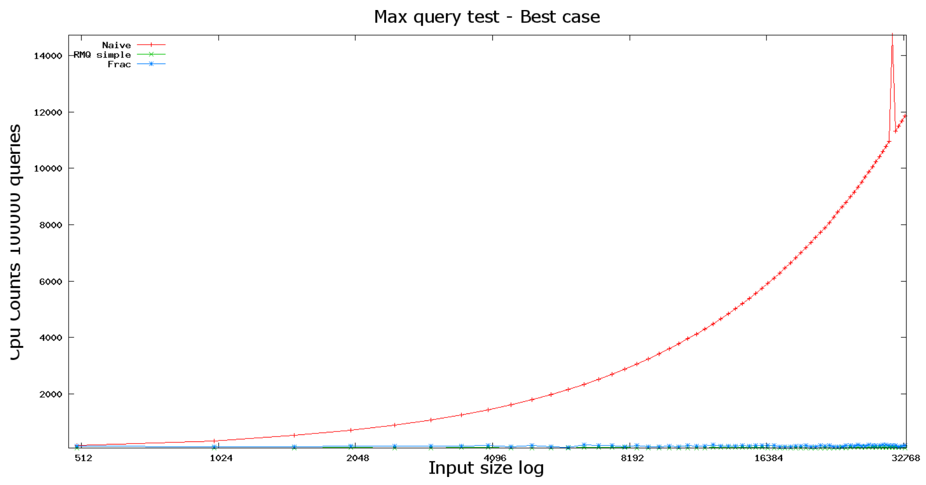

Figure 22: Query time of all the algorithms, best case.

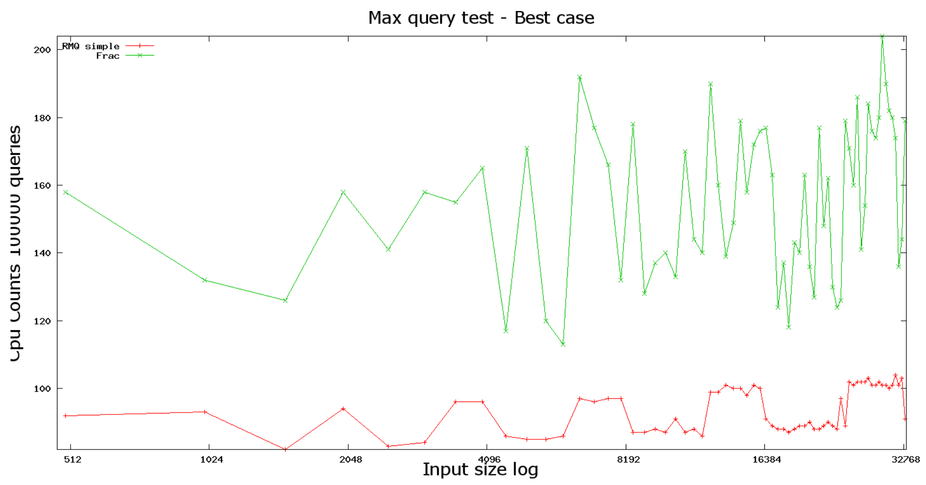

Figure 23: Query time of RMQ simple and Fractional algorithms, best case.

Figure 24 and figure 25 shows RMQ simpel and Fractional both divided by $\log n$. Both figures shows that the algorithms are faster than the $\log n$ boundary. So we can conclude that the absence of querying subtrees have more than a constant effect on the query time, especially in the RMQ simpel case. It were not expected to be better than $\log n$ because there were still a binary tree to query through in both cases, but this could be because of caching.

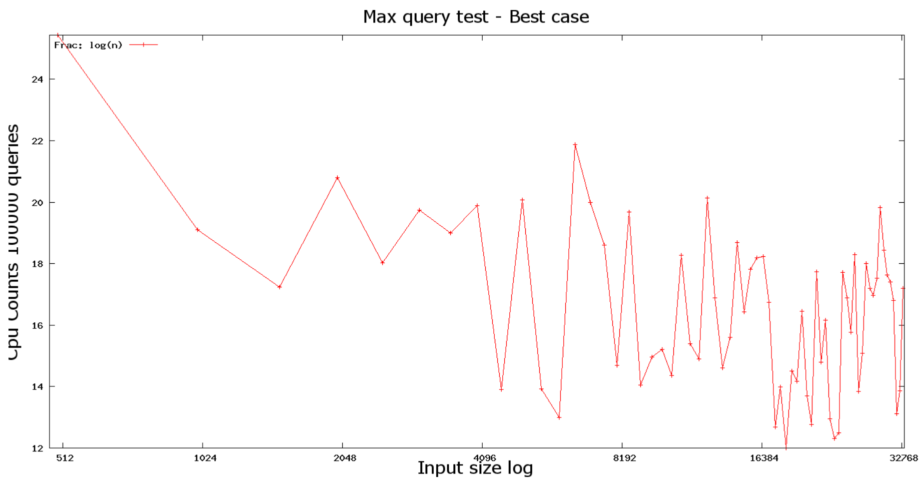

Figure 24: Fractiona divided by $\log n$, best case.

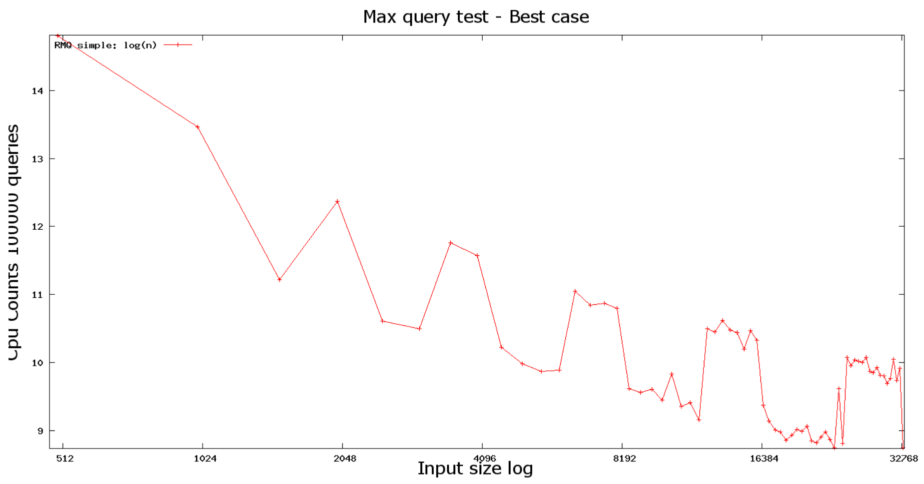

Figure 25: RMQ simple divided by $\log n$, best case.

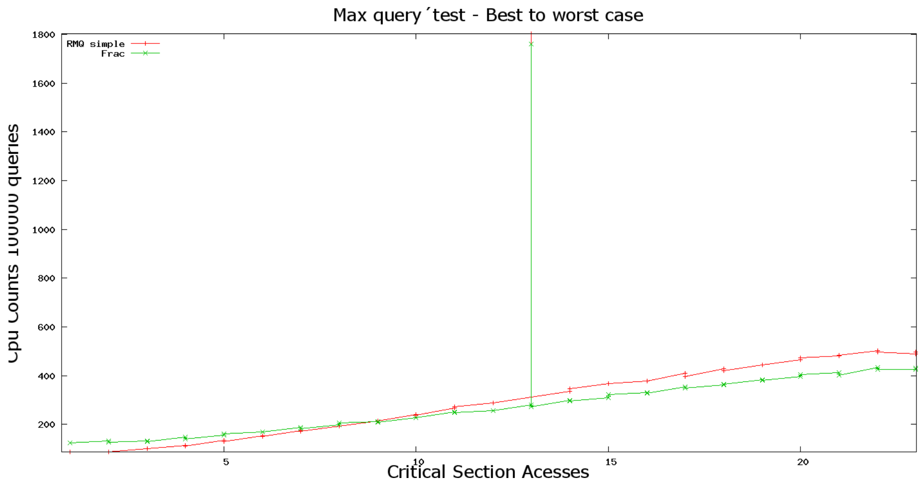

Figure 26: Query time of RMQ simple and Fractional algorithms based on critical sections, best to worst case.

9.6 Best to worst case

The purpose of this test is to measure when Fractional will be faster than RMQ simple, with a fixed size case that starts at the best case and turns more to worst case at every iteration. The input point set is generated using the "generateBestToWorstCase" pseudo code, where the amount of points on the skyline can be controlled if the max query is used. The input for the orthogonal range skyline counting query range is two points that spans the whole input point set, the max query. There will be an individual test for each algorithm. The input set size is fixed at 8192 and at every iteration the amount of points on the skyline changes, so that it will trigger one more sub tree. This is done by using the pseudo code "exampleOfUse" to generate the points to iterate over in combination with 100000 identical max queries.

The expectation is that every second iteration will add one more call to a subtree. This will result in the test starting in the best case and at every iteration changing more to the worst case. This is done to see how many sub trees that needs to be activated, critical sections, before Fractional is better than RMQ simple. Because the critical sections is the main difference between RMQ simple and Fractional queries, then this is also the primary reason for the difference in query times relative to each other.

Figure 26 shows that Fractional becomes faster around 11 critical sections, so that is around half of the possible critical sections, 23, but this is only the smaller critical sections. The first 11 section covers 1024 points total out of 8190 at maximum. The amount of points in the sub trees activated could have a big effect on RMQ simple, because its sub query time is dependent on the amount of points in the subtree, and Fractional is not. This will be tested in the last test section.

There is an anomaly around 13 critical sections and this always shows up in the 24th iteration when testing RMQ simple and Fractional, but not Naive. This is no matter where in the tree structure this 24 iteration activates critical sections, so this is very likely some test specific bug that would no be found.

9.7 Variable query regions

All the tests up to now were max query regions, where every point were included in the query, in this test the effect of smaller query sizes is explored. In this test the input set size is constant at 40000 and generated with the "generateWorstCase" code. There is four tests that all test each algorithm. These tests starts with a max query and at every iteration will make the orthogonal range more narrow by moving one of the sides. So there is a test were the top, bottom, left and right side is moved for each of the three algorithms. At every iteration the side is changed by a 1000, because the test uses "generateWorstCase" then this will result in 1000 points being excluded at every iteration, expect the first and last 1000 points there the distance of each iteration will be binary increasing or decreasing, starting at 1 and ending at 1024. This is done to catch the big difference in critical sections at the beginning and end of the test.

The expectation is that the fewer point that gets queried the fewer critical sections there will be, resulting in a faster query time. Beside this their will probably bee a bigger performance boost over the test in the naive case than at the previous tests, when it were a best case. The difference is that in this case the Naive solution does not need to iterate over all the points, because they are not included in the query anymore, this is a better performance boost than just not needing to add the point to the skyline count.

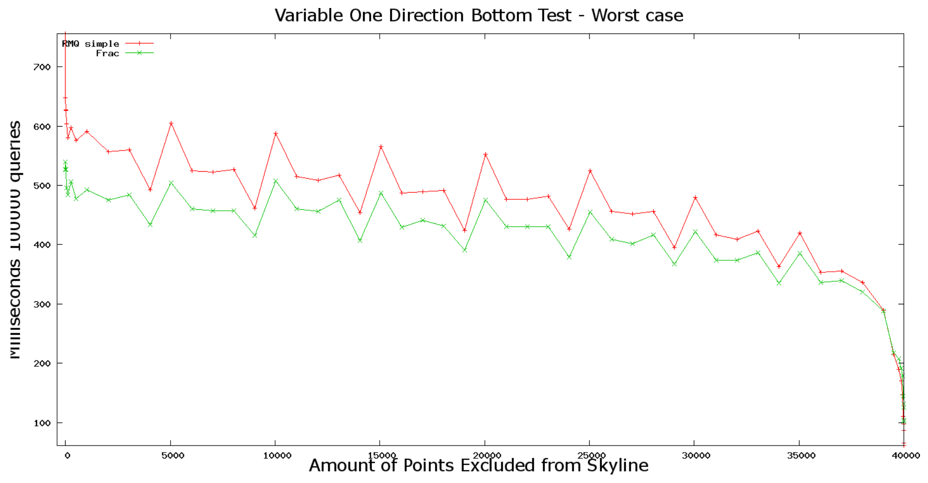

Figure 27: RMQ simple and Fractional query times in the Bottom test, Variable query test.

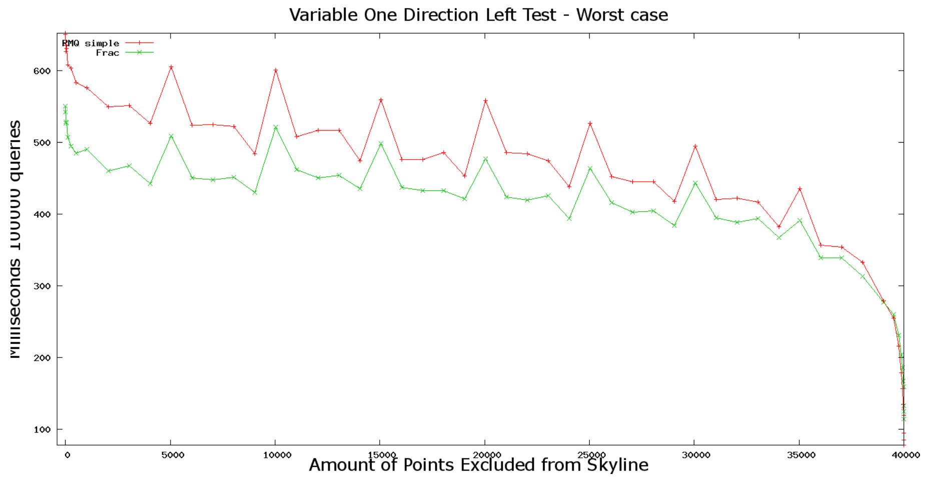

Figure 28: RMQ simple and Fractional query times in the Left test, Variable query test.

Figure 27, 28, 29 and 30 is very similar, because they do have the same amount of points that is queried at every iteration, primarily because of the unique structure of the input set. This test combined with the best to worst case test shows that what matter is the amount of critical sections in the query and the size and placement of the query only matters in relation to how many critical sections it activates.

Figure 31 shows one of the tests together with the Naive algorithm, all the tests give the same result, so only one were shown. There is a drastic decline in the query time of Naive through this test, at the last 250 points it even becomes faster than the two other algorithms.

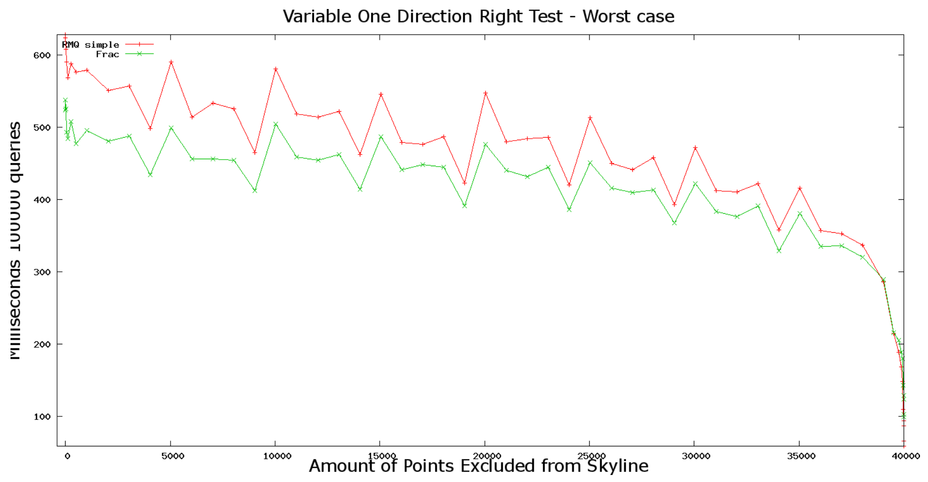

Figure 29: RMQ simple and Fractional query times in the Right test, Variable query test.

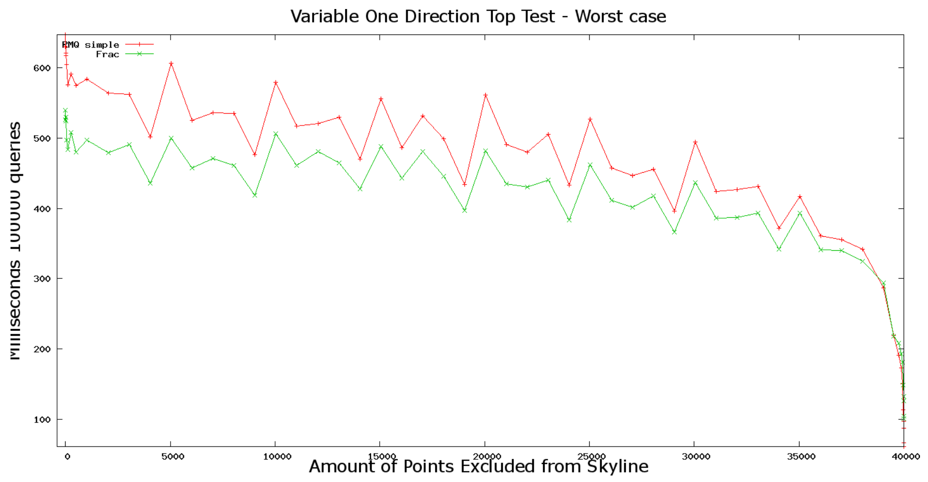

Figure 30: RMQ simple and Fractional query times in the Top test, Variable query test.

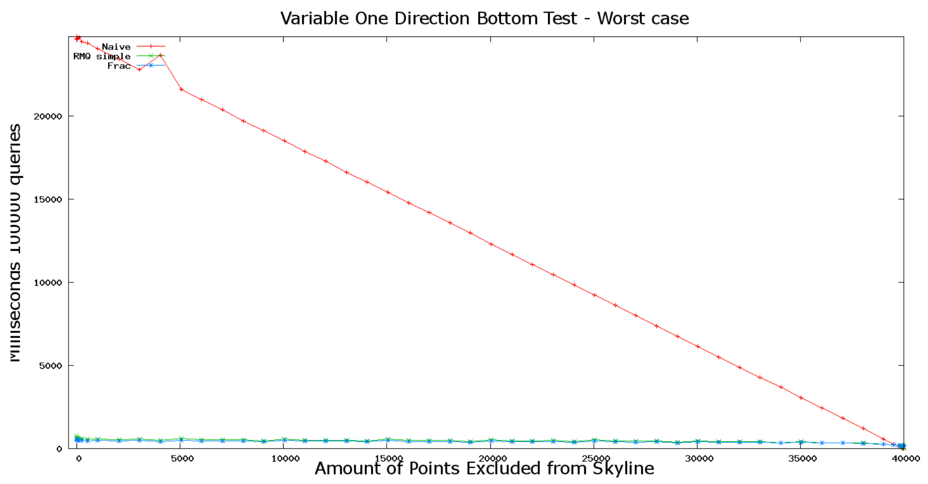

Figure 31: All the algorithms query time in the Bottom test, Variable query test.

9.8 Random case

The purpose of this test is to analyze random cases and how Fractional and RMQ simple queries perform in these cases. The current expectation is that the running time of the query is primarily dependent on the amount of critical sections accesses and the amount of points the sections span over. It is also expected that Fractional is better than RMQ simple when there are more critical sections accesses.

The input set size, *size*, is fixed at 40000 points, every point in the input set is generated randomly where $0 \leq x \leq size$ and $0 \leq y \leq size$. The input for the orthogonal range skyline counting query range is two points that is also generated randomly where the coordinates for the lower left is $0 \leq x < size$ and $0 \leq y < size$ and the top right is $lowerLeft.x < x \leq size$ and $lowerLeft.y < y \leq size$, to always create a valid query range.

There is 300 iterations during the test, where every iteration generate a new random input set and for both algorithms a new structure based on this input. For every iteration there are 50 sub iterations where at every sub iteration there is generated new random orthogonal range points and performed 100000 identical queries with that range.

This will result in many measurements with the same amount of critical sections accesses but different query times, because the size of the critical section that got accessed have an impact especially on RMQ simple.

Figure 32 shows the distribution of cases over amount of critical sections accesses for Fractional and RMQ simple. First thing to note is that out of $\lfloor \log 40000 \rfloor * 2 = 30$ possible critical section accesses then only 14 gets accessed and in most cases only 4 or 5, so the difference between Fractional and RMQ simple in this test will probably seem smaller than in the previous tests. Second thing is because of the normal distribution of the cases then the data from the extreme high and low cases there is only a couple of measurements so it probably wont be as trustworthy. A last thing to note is that Fractional have fewer critical section accesses and this is because there were implemented more checks before a critical section in Fractional than it was practical in RMQ simple.

Figure 33, 34 and 35 shows the minimum, maximum and average query time for each algorithm over the amount of critical section accesses cases.

In the maximum figure it shows that even at one critical section it is possible for Fractional to be faster than RMQ Simple, this is probably because that one section were very big and this can have a big impact on the RMQ simples query time. Another thing to note is that the query time do not rise with the critical section accesses, it seems like the first couple of big critical sections uses the majority of the time and compared to them the rest does not matter.

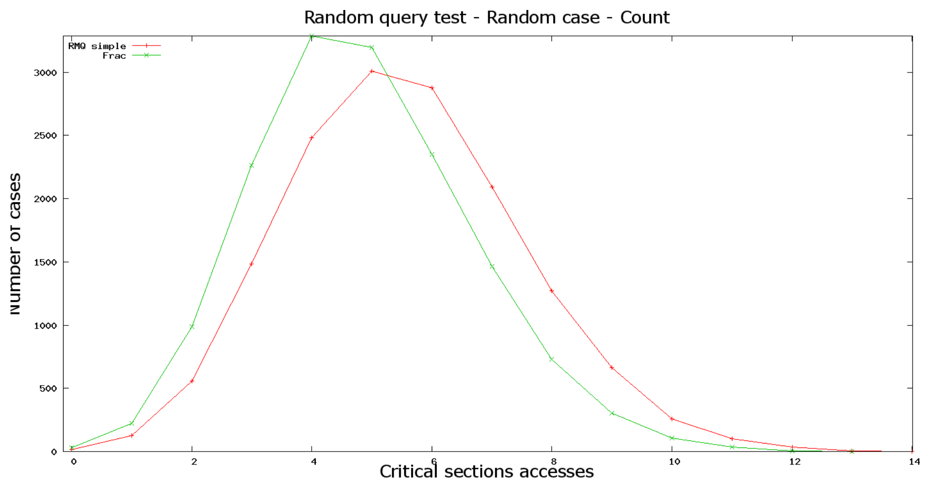

Figure 32: RMQ simple and Fractional distribution of cases over amount of critical sections accesses, random test.

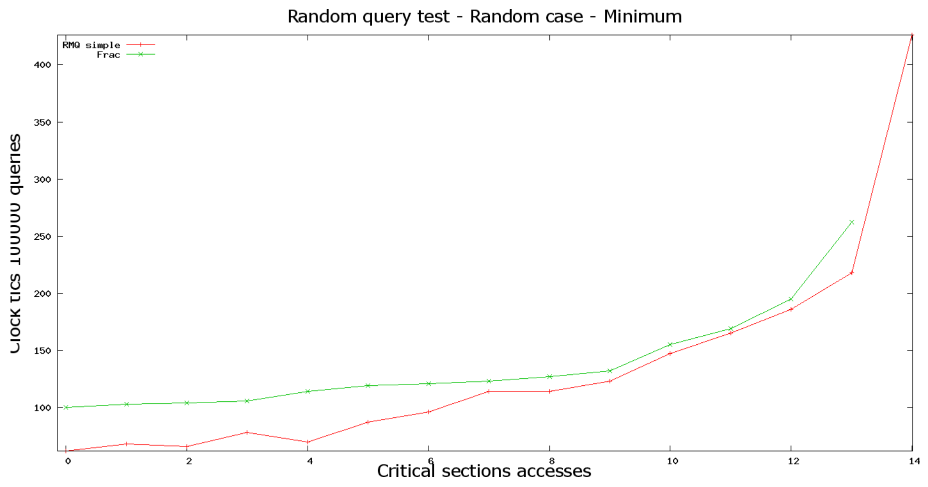

Figure 33: RMQ simple and Fractional minimum query times, random test.

In the minimum figure it shows that Fractional is slower than RMQ simple, this is probably because if all the critical sections spans very few elements, then the bigger structure of Fractional outweighs the small amounts of time in the binary search of RMQ simples sub queries.

In the average figure it shows that Fractional quickly becomes faster than RMQ simple and it seems to be a steady development. This happens earlier than in the best to worst case test, but it is probably because the critical section that spans a lot of points is both expensive and have a large risk to be queried, where in the best to worst case the test started with the critical section that spanned few points.

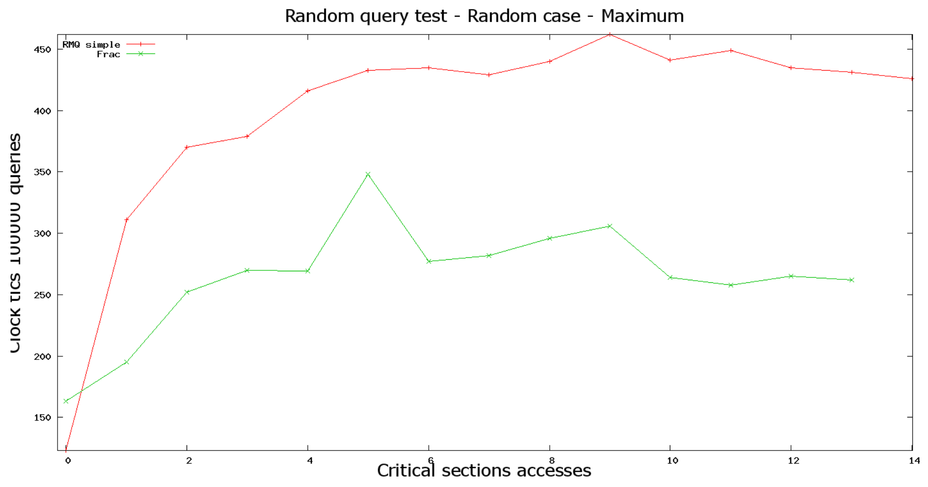

Figure 34: RMQ simple and Fractional maximum query times, random test.

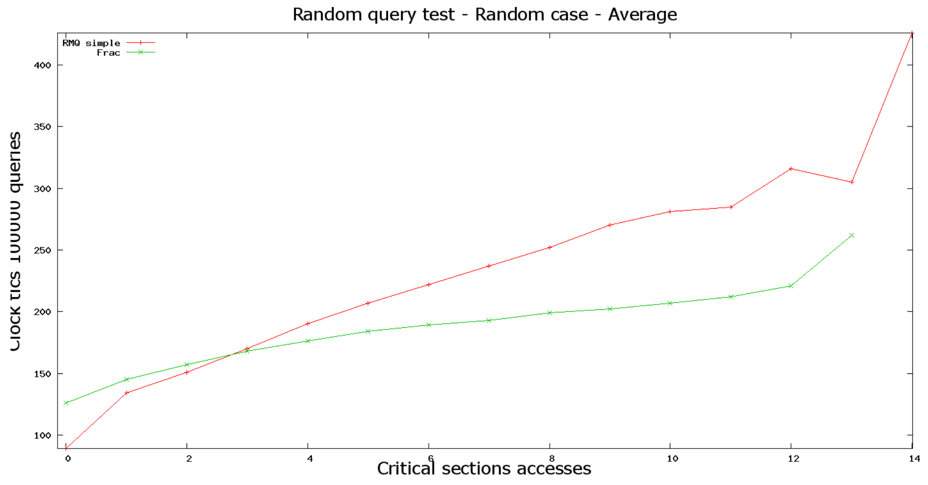

Figure 35: RMQ simple and Fractional average query times, random test.

10 Conclusion

The topic of this thesis were to implement and test these steps up to the final structure:

1. Naive: $O(n)$ query time, $O(n \log n)$ preprocessing and $O(n)$ words space usage. Iteration over a sorted list.

2. RMQ simple: $O(\log^2 n)$ query time, $O(n \log^2 n)$ preprocessing and words space usage. Introducing a binary tree structure and non succinct range maximum structure [Bender and Farach-Colton, 2000] and a variation on non succinct dominating prefix sum [Brodal and Larsen, 2014] called non succinct prefix skyline count.

3. Fractional: $O(\log n)$ query time, $O(n \log^2 n)$ preprocessing and words space usage. Introducing a non succinct fractional cascading predecessor and successor structure [Chazelle and Guibas, 1986].

4. Succinct RMQ: $O(\log n)$ query time, $O(n \log n)$ preprocessing and words space usage. Introducing a succinct range maximum structure [Fischer, 2008].

5. Succinct dominating prefix sum and fractional cascading: $O(\log n)$ query time, $O(n \log n)$ preprocessing and $O(n)$ words space usage. Introducing a succinct dominating prefix sum structure and a succinct fractional cascading structure, using [Raman et al., 2007].

6. Final Structure: $O(\log n / \log \log n)$ query time, $O(n \log n)$ preprocessing and $O(n)$ words space usage. Introducing a delta tree structure [Brodal and Larsen, 2014] [Das et al., 2013] and fractional predecessor and successor structure [Brodal and Larsen, 2014].

The purpose were to test the different steps and compare the performance to each other. During the paper the first three steps were implemented and tested. Detailed description of the theory and implementation of the steps, test settings and test results is found in this paper. The third step implemented the algorithm with the best query time, for a binary tree, but without the succinct optimizations to the sub structures so it could achieve best space usage, for a binary tree. The theory for the rest of the steps are described in this paper too.

The paper concludes through tests that the algorithms follow the query times, preprocessing times and structure size boundaries set in the theories. There is also an analysis of the best case scenario where RMQ simple performed better than Fractional, even though Fractional where better in the worst case. This got followed by a best to worst case scenario test that showed RMQ simple quickly became worse than Fractional. At the end there were a random test that showed RMQ simple in many random cases were slower than Fractional, because of the high risk of querying a big subtree in the binary structure.

It could have been interesting to see how the implementation of the succinct structures would have effected the query time, and if the very complex final structures many lookups and calculations would have made it slower in practice, than the binary structures. But this is work for the future.

11 References

[Bender and Farach-Colton, 2000] Bender, M. A. and Farach-Colton, M. (2000). The lca problem revisited. In Gonnet, G. H., Panario, D., and Viola, A., editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer. All chapters.

[Brodal and Larsen, 2014] Brodal, G. S. and Larsen, K. G. (2014). Optimal planar orthogonal skyline counting queries. In *Proc. 14th Scandinavian Workshop on Algorithm Theory*, volume 8503 of *Lecture Notes in Computer Science*, pages 98–109. Springer Verlag, Berlin. Chapter 1 and 3.

[Chazelle and Guibas, 1986] Chazelle, B. and Guibas, L. J. (1986). Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162. Chapter 1.

[Das et al., 2012] Das, A. S., Gupta, P., Kalavagattu, A. K., Agarwal, J., Srinathan, K., and Kothapalli, K. (2012). Range aggregate maximal points in the plane. In Rahman, M. S. and Nakano, S.-I., editors, *WALCOM*, volume 7157 of *Lecture Notes in Computer Science*, pages 52–63. Springer. Chapter 1.

[Das et al., 2013] Das, A. S., Gupta, P., and Srinathan, K. (2013). Counting maximal points in a query orthogonal rectangle. In Ghosh, S. K. and Tokuyama, T., editors, *WALCOM*, volume 7748 of *Lecture Notes in Computer Science*, pages 65–76. Springer. Chapter 1.

[Fischer, 2008] Fischer, J. (2008). Optimal succinctness for range minimum queries. *CoRR*, abs/0812.2775. Chapter 2-3.

[JáJá et al., 2004] JáJá, J., Mortensen, C. W., and Shi, Q. (2004). Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Fleischer, R. and Trippen, G., editors, *ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer. Chapter 1.

[Kalavagattu et al., 2012] Kalavagattu, A. K., Agarwal, J., Das, A. S., and Kothapalli, K. (2012). On counting range maxima points in plane. In Arumugam, S. and Smyth, W. F., editors, *IWOCA*, volume 7643 of *Lecture Notes in Computer Science*, pages 263–273. Springer. Chapter 1.

[Patrascu, 2007] Patrascu, M. (2007). Lower bounds for 2-dimensional range counting. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 40–46, New York, NY, USA. ACM. Chapter 1.

[Raman et al., 2002] Raman, R., Raman, V., and Rao, S. S. (2002). Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 233–242, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics. Chapter 1-2.1.

[Raman et al., 2007] Raman, R., Raman, V., and Satti, S. R. (2007). Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4). Chapter 1-2.1.

12 Apendixes A: Table of notation

S	Set of points.
S	Subset of points $S' \subset S$.
n	Input size.
p	A point.
P	Set of points.
log	Binary log.
A	Sorted array of points.
RMQ	Range minimum query.
LCA	Least common ancestor.
$Xmax$	The leaf containing the point with the maximum x value in the query range.
$Xmin$	The leaf containing the point with the minimum x value in the query range.
low x path	The path from $Xmax$ to the LCA.
high x path	The path from $Xmin$ to the LCA.
highSky	The point on the skyline with the highest y coordinate.
lowSky	The point on the skyline with the lowest y coordinate.
sortedY	The points an inner node spans, sorted with respect to the y axis, Definition 1.4.
lowY	The lowest y value a point can have and still be considered in a sub query.
highY	The highest y value a point can have and still be considered in a sub query.
lowerLeft	The lower left point of the orthogonal range.
topRight	The top right point of the orthogonal range.
RMQ	Range maximum query, Lemma 4.5 and 6.2.
RMQ simple	The orthogonal range skyline counting query defined by Lemma 4.7.
Fractional	Can refer to the orthogonal range skyline counting query defined by Lemma 5.8.
Naive	The orthogonal range skyline counting query defined by Lemma 3.3.