answer the queries in $O(\log^2 n)$ time, we just have to make a binary search on the second coordinate, but this will not worsen the complexity.

But modification queries will be impossible with this structure: in fact if a new point appears, we have to add a new element in the middle of some Segment Tree along the second coordinate, which cannot be effectively done.

In conclusion we note that the two-dimensional Segment Tree contracted in the described way becomes practically equivalent to the modification of the one-dimensional Segment Tree (see Saving the entire subarrays in each vertex). In particular the two-dimensional Segment Tree is just a special case of storing a subarray in each vertex of the tree. It follows, that if you gave to abandon a two-dimensional Segment Tree due to the impossibility of executing a query, it makes sense to try to replace the nested Segment Tree with some more powerful data structure, for example a Cartesian tree.

Preserving the history of its values (Persistent Segment Tree)

A persistent data structure is a data structure that remembers it previous state for each modification. This allows to access any version of this data structure that interest us and execute a query on it.

Segment Tree is a data structure that can be turned into a persistent data structure efficiently (both in time and memory consumption). We want to avoid copying the complete tree before each modification, and we don't want to loose the $O(\log n)$ time behavior for answering range queries.

In fact, any change request in the Segment Tree leads to a change in the data of only $O(\log n)$ vertices along the path starting from the root. So if we store the Segment Tree using pointers (i.e. a vertex stores pointers to the left and the right child vertices), then when performing the modification query, we simply need to create new vertices instead of changing the available vertices. Vertices that are not affected by the modification query can still be used by pointing the pointers to the old vertices. Thus for a modification query $O(\log n)$ new vertices will be created, including a new root vertex of the Segment Tree, and the entire previous version of the tree rooted at the old root vertex will remain unchanged.

Let's give an example implementation for the simplest Segment Tree: when there is only a query asking for sums, and modification queries of single elements.

```
struct Vertex {
    Vertex *l, *r;
    int sum;

    Vertex(int val) : l(nullptr), r(nullptr), sum(val) {}
    Vertex(Vertex *l, Vertex *r) : l(l), r(r), sum(0) {
        if (l) sum += l->sum;
        if (r) sum += r->sum;
    }
};

Vertex* build(int a[], int tl, int tr) {
    if (tl == tr)
        return new Vertex(a[tl]);
    int tm = (tl + tr) / 2;
    return new Vertex(build(a, tl, tm), build(a, tm+1, tr));
}

int get_sum(Vertex* v, int tl, int tr, int l, int r) {
```

```
    if (l > r)
        return 0;
    if (l == tl && tr == r)
        return v->sum;
    int tm = (tl + tr) / 2;
    return get_sum(v->l, tl, tm, l, min(r, tm))
         + get_sum(v->r, tm+1, tr, max(l, tm+1), r);
}

Vertex* update(Vertex* v, int tl, int tr, int pos, int new_val) {
    if (tl == tr)
        return new Vertex(new_val);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return new Vertex(update(v->l, tl, tm, pos, new_val), v->r);
    else
        return new Vertex(v->l, update(v->r, tm+1, tr, pos, new_val));
}
```

For each modification of the Segment Tree we will receive a new root vertex. To quickly jump between two different versions of the Segment Tree, we need to store this roots in an array. To use a specific version of the Segment Tree we simply call the query using the appropriate root vertex.

With the approach described above almost any Segment Tree can be turned into a persistent data structure.

Finding the k-th smallest number in a range

This time we have to answer queries of the form "What is the k-th smallest element in the range $a[l \dots r]$. This query can be answered using a binary search and a Merge Sort Tree, but the time complexity for a single query would be $O(\log^3 n)$. We will accomplish the same task using a persistent Segment Tree in $O(\log n)$.

First we will discuss a solution for a simpler problem: We will only consider arrays in which the elements are bound by $0 \le a[i] < n$. And we only want to find the k-th smallest element in some prefix of the array a. It will be very easy to extent the developed ideas later for not restricted arrays and not restricted range queries. Note that we will be using 1 based indexing for a.

We will use a Segment Tree that counts all appearing numbers, i.e. in the Segment Tree we will store the histogram of the array. So the leaf vertices will store how often the values 0, 1, ..., $n-1$ will appear in the array, and the other vertices store how many numbers in some range are in the array. In other words we create a regular Segment Tree with sum queries over the histogram of the array. But instead of creating all n Segment Trees for every possible prefix, we will create one persistent one, that will contain the same information. We will start with an empty Segment Tree (all counts will be 0) pointed to by $root_0$, and add the elements $a[1]$, $a[2]$, ..., $a[n]$ one after another. For each modification we will receive a new root vertex, let's call $root_i$ the root of the Segment Tree after inserting the first i elements of the array a. The Segment Tree rooted at $root_i$ will contain the histogram of the prefix $a[1 \dots i]$. Using this Segment Tree we can find in $O(\log n)$ time the position of the k-th element using the same technique discussed in Counting the number of zeros, searching for the k-th zero.

Now to the not-restricted version of the problem.

First for the restriction on the queries: Instead of only performing these queries over a prefix of a, we want to use any arbitrary segments $a[l \dots r]$. Here we need a Segment Tree that represents the

histogram of the elements in the range $a[l \dots r]$. It is easy to see that such a Segment Tree is just the difference between the Segment Tree rooted at $root_r$ and the Segment Tree rooted at $root_{l-1}$, i.e. every vertex in the $[l \dots r]$ Segment Tree can be computed with the vertex of the $root_r$ tree minus the vertex of the $root_{l-1}$ tree.

In the implementation of the find_kth function this can be handled by passing two vertex pointer and computing the count/sum of the current segment as difference of the two counts/sums of the vertices.

Here are the modified build, update and find_kth functions

```
Vertex* build(int tl, int tr) {
    if (tl == tr)
        return new Vertex(0);
    int tm = (tl + tr) / 2;
    return new Vertex(build(tl, tm), build(tm+1, tr));
}

Vertex* update(Vertex* v, int tl, int tr, int pos) {
    if (tl == tr)
        return new Vertex(v->sum+1);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return new Vertex(update(v->l, tl, tm, pos), v->r);
    else
        return new Vertex(v->l, update(v->r, tm+1, tr, pos));
}

int find_kth(Vertex* vl, Vertex *vr, int tl, int tr, int k) {
    if (tl == tr)
        return tl;
    int tm = (tl + tr) / 2, left_count = vr->l->sum - vl->l->sum;
    if (left_count >= k)
        return find_kth(vl->l, vr->l, tl, tm, k);
    return find_kth(vl->r, vr->r, tm+1, tr, k-left_count);
}
```

As already written above, we need to store the root of the initial Segment Tree, and also all the roots after each update. Here is the code for building a persistent Segment Tree over an vector `a` with elements in the range `[0, MAX_VALUE]`.

```
int tl = 0, tr = MAX_VALUE + 1;
std::vector<Vertex*> roots;
roots.push_back(build(tl, tr));
for (int i = 0; i < a.size(); i++) {
    roots.push_back(update(roots.back(), tl, tr, a[i]));
}

// find the 5th smallest number from the subarray [a[2], a[3], ..., a[19]]
int result = find_kth(roots[2], roots[20], tl, tr, 5);
```

Now to the restrictions on the array elements: We can actually transform any array to such an array by index compression. The smallest element in the array will gets assigned the value 0, the second smallest the value 1, and so forth. It is easy to generate lookup tables (e.g. using map), that convert a value to its index and vice versa in $O(\log n)$ time.

Dynamic segment tree

(Called so because its shape is dynamic and the nodes are usually dynamically allocated. Also known as *implicit segment tree* or *sparse segment tree*.)

Previously, we considered cases when we have the ability to build the original segment tree. But what to do if the original size is filled with some default element, but its size does not allow you to completely build up to it in advance?

We can solve this problem by creating a segment tree lazily (incrementally). Initially, we will create only the root, and we will create the other vertexes only when we need them. In this case, we will use the implementation on pointers(before going to the vertex children, check whether they are created, and if not, create them). Each query has still only the complexity $O(\log n)$, which is small enough for most use-cases (e.g. $\log_2 10^9 \approx 30$).

In this implementation we have two queries, adding a value to a position (initially all values are 0), and computing the sum of all values in a range. `Vertex(0, n)` will be the root vertex of the implicit tree.

```
struct Vertex {
    int left, right;
    int sum = 0;
    Vertex *left_child = nullptr, *right_child = nullptr;

    Vertex(int lb, int rb) {
        left = lb;
        right = rb;
    }

    void extend() {
        if (!left_child && left + 1 < right) {
            int t = (left + right) / 2;
            left_child = new Vertex(left, t);
            right_child = new Vertex(t, right);
        }
    }

    void add(int k, int x) {
        extend();
        sum += x;
        if (left_child) {
            if (k < left_child->right)
                left_child->add(k, x);
            else
                right_child->add(k, x);
        }
    }

    int get_sum(int lq, int rq) {
        if (lq <= left && right <= rq)
            return sum;
        if (max(left, lq) >= min(right, rq))
            return 0;
        extend();
        return left_child->get_sum(lq, rq) + right_child->get_sum(lq, rq);
    }
};
```