Sqrt Tree

Table of Contents

Given an array a that contains n elements and the operation $\circ$ that satisfies associative property: $(x \circ y) \circ z = x \circ (y \circ z)$ is true for any x, y, z.

So, such operations as $\gcd$, $\min$, $\max$, $+$, and, or, xor, etc. satisfy these conditions.

Also we have some queries $q(l, r)$. For each query, we need to compute $a_l \circ a_{l+1} \circ \cdots \circ a_r$.

Sqrt Tree can process such queries in $O(1)$ time with $O(n \cdot \log \log n)$ preprocessing time and $O(n \cdot \log \log n)$ memory.

Description

Building sqrt decomposition

Let's make a sqrt decomposition. We divide our array in $\sqrt{n}$ blocks, each block has size $\sqrt{n}$. For each block, we compute:

1. Answers to the queries that lie in the block and begin at the beginning of the block ($\mathrm{prefixOp}$)
2. Answers to the queries that lie in the block and end at the end of the block ($\mathrm{suffixOp}$)

And we'll compute an additional array:

1. $\mathrm{between}_{i,j}$ (for $i \le j$) - answer to the query that begins at the start of block i and ends at the end of block j. Note that we have $\sqrt{n}$ blocks, so the size of this array will be $O(\sqrt{n}^2) = O(n)$.

Let's see the example.

Let $\circ$ be $+$ (we calculate sum on a segment) and we have the following array a:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

It will be divided onto three blocks: `{1, 2, 3}`, `{4, 5, 6}` and `{7, 8, 9}`.

For first block $\mathrm{prefixOp}$ is `{1, 3, 6}` and $\mathrm{suffixOp}$ is `{6, 5, 3}`.

For second block $\mathrm{prefixOp}$ is `{4, 9, 15}` and $\mathrm{suffixOp}$ is `{15, 11, 6}`.

For third block $\mathrm{prefixOp}$ is `{7, 15, 24}` and $\mathrm{suffixOp}$ is `{24, 17, 9}`.

$\mathrm{between}$ array is:

```
{
    {6, 21, 45},
    {0, 15, 39},
    {0, 0,  24}
}
```

(we assume that invalid elements where $i > j$ are filled with zeroes)

It's obvious to see that these arrays can be easily calculated in $O(n)$ time and memory.

We already can answer some queries using these arrays. If the query doesn't fit into one block, we can divide it onto three parts: suffix of a block, then some segment of contiguous blocks and then prefix of some block. We can answer a query by dividing it into three parts and taking our operation of some value from $\mathrm{suffixOp}$, then some value from $\mathrm{between}$, then some value from $\mathrm{prefixOp}$.

But if we have queries that entirely fit into one block, we cannot process them using these three arrays. So, we need to do something.

Making a tree

We cannot answer only the queries that entirely fit in one block. But what **if we build the same structure as described above for each block?** Yes, we can do it. And we do it recursively, until we reach the block size of 1 or 2. Answers for such blocks can be calculated easily in $O(1)$.

So, we get a tree. Each node of the tree represents some segment of the array. Node that represents array segment with size k has $\sqrt{k}$ children -- for each block. Also each node contains the three arrays described above for the segment it contains. The root of the tree represents the entire array. Nodes with segment lengths 1 or 2 are leaves.

Also it's obvious that the height of this tree is $O(\log \log n)$, because if some vertex of the tree represents an array with length k, then its children have length $\sqrt{k}$. $\log(\sqrt{k}) = \frac{\log k}{2}$, so $\log k$ decreases two times every layer of the tree and so its height is $O(\log \log n)$. The time for building and memory usage will be $O(n \cdot \log \log n)$, because every element of the array appears exactly once on each layer of the tree.

Now we can answer the queries in $O(\log \log n)$. We can go down on the tree until we meet a segment with length 1 or 2 (answer for it can be calculated in $O(1)$ time) or meet the first segment in which our query doesn't fit entirely into one block. See the first section on how to answer the query in this case.

OK, now we can do $O(\log \log n)$ per query. Can it be done faster?

Optimizing the query complexity

One of the most obvious optimization is to binary search the tree node we need. Using binary search, we can reach the $O(\log \log \log n)$ complexity per query. Can we do it even faster?

The answer is yes. Let's assume the following two things:

1. Each block size is a power of two.
2. All the blocks are equal on each layer.

To reach this, we can add some zero elements to our array so that its size becomes a power of two.

When we use this, some block sizes may become twice larger to be a power of two, but it still be $O(\sqrt{k})$ in size and we keep linear complexity for building the arrays in a segment.

Now, we can easily check if the query fits entirely into a block with size 2^k. Let's write the ranges of the query, l and r (we use 0-indexation) in binary form. For instance, let's assume $k = 4, l = 39, r = 46$. The binary representation of l and r is:

$$l = 39_{10} = 100111_2$$

$$r = 46_{10} = 101110_2$$

Remember that one layer contains segments of the equal size, and the block on one layer have also equal size (in our case, their size is $2^k = 2^4 = 16$. The blocks cover the array entirely, so the first block covers elements $(0 - 15)$ ($(000000_2 - 001111_2)$ in binary), the second one covers elements $(16 - 31)$ ($(010000_2 - 011111_2)$ in binary) and so on. We see that the indices of the positions covered by one block may differ only in k (in our case, 4) last bits. In our case l and r have equal bits except four lowest, so they lie in one block.

So, we need to check if nothing more that k smallest bits differ (or $l \operatorname{xor} r$ doesn't exceed $2^k - 1$).

Using this observation, we can find a layer that is suitable to answer the query quickly. How to do this:

1. For each i that doesn't exceed the array size, we find the highest bit that is equal to 1. To do this quickly, we use DP and a precalculated array.

2. Now, for each $q(l, r)$ we find the highest bit of $l \operatorname{xor} r$ and, using this information, it's easy to choose the layer on which we can process the query easily. We can also use a precalculated array here.

For more details, see the code below.

So, using this, we can answer the queries in $O(1)$ each. Hooray! :)

Updating elements

We can also update elements in Sqrt Tree. Both single element updates and updates on a segment are supported.

Updating a single element

Consider a query $\mathrm{update}(x, val)$ that does the assignment $a_x = val$. We need to perform this query fast enough.

Naive approach

First, let's take a look of what is changed in the tree when a single element changes. Consider a tree node with length l and its arrays: $\mathrm{prefixOp}$, $\mathrm{suffixOp}$ and $\mathrm{between}$. It is easy to see that only $O(\sqrt{l})$ elements from $\mathrm{prefixOp}$ and $\mathrm{suffixOp}$ change (only inside the block with the changed element). $O(l)$ elements are changed in $\mathrm{between}$. Therefore, $O(l)$ elements in the tree node are updated.

We remember that any element x is present in exactly one tree node at each layer. Root node (layer 0) has length $O(n)$, nodes on layer 1 have length $O(\sqrt{n})$, nodes on layer 2 have length $O(\sqrt{\sqrt{n}})$, etc. So the time complexity per update is $O(n + \sqrt{n} + \sqrt{\sqrt{n}} + \ldots) = O(n)$.

But it's too slow. Can it be done faster?

An sqrt-tree inside the sqrt-tree

Note that the bottleneck of updating is rebuilding $\mathrm{between}$ of the root node. To optimize the tree, let's get rid of this array! Instead of $\mathrm{between}$ array, we store another sqrt-tree for the root node. Let's call it index. It plays the same role as $\mathrm{between}$— answers the queries on segments of blocks. Note that the rest of the tree nodes don't have index, they keep their $\mathrm{between}$ arrays.

A sqrt-tree is *indexed*, if its root node has index. A sqrt-tree with $\mathrm{between}$ array in its root node is *unindexed*. Note that index **is *unindexed* itself**.

So, we have the following algorithm for updating an *indexed* tree:

- Update $\mathrm{prefixOp}$ and $\mathrm{suffixOp}$ in $O(\sqrt{n})$.
- Update index. It has length $O(\sqrt{n})$ and we need to update only one item in it (that represents the changed block). So, the time complexity for this step is $O(\sqrt{n})$. We can use the algorithm described in the beginning of this section (the "slow" one) to do it.
- Go into the child node that represents the changed block and update it in $O(\sqrt{n})$ with the "slow" algorithm.

Note that the query complexity is still $O(1)$: we need to use index in query no more than once, and this will take $O(1)$ time.

So, total time complexity for updating a single element is $O(\sqrt{n})$. Hooray! :)

Updating a segment

Sqrt-tree also can do things like assigning an element on a segment. $\mathrm{massUpdate}(x, l, r)$ means $a_i = x$ for all $l \le i \le r$.

There are two approaches to do this: one of them does $\mathrm{massUpdate}$ in $O(\sqrt{n} \cdot \log\log n)$, keeping $O(1)$ per query. The second one does $\mathrm{massUpdate}$ in $O(\sqrt{n})$, but the query complexity becomes $O(\log\log n)$.

We will do lazy propagation in the same way as it is done in segment trees: we mark some nodes as *lazy*, meaning that we'll push them when it's necessary. But one thing is different from segment trees: pushing a node is expensive, so it cannot be done in queries. On the layer 0, pushing a node takes $O(\sqrt{n})$ time. So, we don't push nodes inside queries, we only look if the current node or its parent are *lazy*, and just take it into account while performing queries.

First approach

In the first approach, we say that only nodes on layer 1 (with length $O(\sqrt{n})$ can be *lazy*. When pushing such node, it updates all its subtree including itself in $O(\sqrt{n} \cdot \log\log n)$. The $\mathrm{massUpdate}$ process is done as follows:

- Consider the nodes on layer 1 and blocks corresponding to them.
- Some blocks are entirely covered by $\mathrm{massUpdate}$. Mark them as *lazy* in $O(\sqrt{n})$.
- Some blocks are partially covered. Note there are no more than two blocks of this kind. Rebuild them in $O(\sqrt{n} \cdot \log\log n)$. If they were *lazy*, take it into account.
- Update $\mathrm{prefixOp}$ and $\mathrm{suffixOp}$ for partially covered blocks in $O(\sqrt{n})$ (because there are only two such blocks).
- Rebuild the index in $O(\sqrt{n} \cdot \log\log n)$.

So we can do $\mathrm{massUpdate}$ fast. But how lazy propagation affects queries? They will have the following modifications:

- If our query entirely lies in a *lazy* block, calculate it and take *lazy* into account. $O(1)$.
- If our query consists of many blocks, some of which are *lazy*, we need to take care of *lazy* only on the leftmost and the rightmost block. The rest of the blocks are calculated using index, which already knows the answer on *lazy* block (because it's rebuilt after each modification). $O(1)$.

The query complexity still remains $O(1)$.

Second approach

In this approach, each node can be *lazy* (except root). Even nodes in index can be *lazy*. So, while processing a query, we have to look for *lazy* tags in all the parent nodes, i. e. query complexity will be $O(\log \log n)$.

But $\mathrm{massUpdate}$ becomes faster. It looks in the following way:

- Some blocks are fully covered with $\mathrm{massUpdate}$. So, *lazy* tags are added to them. It is $O(\sqrt{n})$.
- Update $\mathrm{prefixOp}$ and $\mathrm{suffixOp}$ for partially covered blocks in $O(\sqrt{n})$ (because there are only two such blocks).
- Do not forget to update the index. It is $O(\sqrt{n})$ (we use the same $\mathrm{massUpdate}$ algorithm).
- Update $\mathrm{between}$ array for *unindexed* subtrees.
- Go into the nodes representing partially covered blocks and call $\mathrm{massUpdate}$ recursively.

Note that when we do the recursive call, we do prefix or suffix $\mathrm{massUpdate}$. But for prefix and suffix updates we can have no more than one partially covered child. So, we visit one node on layer 1, two nodes on layer 2 and two nodes on any deeper level. So, the time complexity is $O(\sqrt{n} + \sqrt{\sqrt{n}} + \ldots) = O(\sqrt{n})$. The approach here is similar to the segment tree mass update.

Implementation

The following implementation of Sqrt Tree can perform the following operations: build in $O(n \cdot \log \log n)$, answer queries in $O(1)$ and update an element in $O(\sqrt{n})$.

```
SqrtTreeItem op(const SqrtTreeItem &a, const S

inline int log2Up(int n) {
    int res = 0;
    while ((1 << res) < n) {
        res++;
    }
    return res;
}

class SqrtTree {
private:
    int n, lg, indexSz;
    vector<SqrtTreeItem> v;
    vector<int> clz, layers, onLayer;
    vector< vector<SqrtTreeItem> > pref, suf,

    inline void buildBlock(int layer, int l, i
        pref[layer][l] = v[l];
        for (int i = l+1; i < r; i++) {
            pref[layer][i] = op(pref[layer][i-
        }
        suf[layer][r-1] = v[r-1];
        for (int i = r-2; i >= l; i--) {
            suf[layer][i] = op(v[i], suf[layer
```

```
    }
}

inline void buildBetween(int layer, int lB
    int bSzLog = (layers[layer]+1) >> 1;
    int bCntLog = layers[layer] >> 1;
    int bSz = 1 << bSzLog;
    int bCnt = (rBound - lBound + bSz - 1)
    for (int i = 0; i < bCnt; i++) {
        SqrtTreeItem ans;
        for (int j = i; j < bCnt; j++) {
            SqrtTreeItem add = suf[layer][
            ans = (i == j) ? add : op(ans,
            between[layer-1][betweenOffs +
        }
    }
}

inline void buildBetweenZero() {
    int bSzLog = (lg+1) >> 1;
    for (int i = 0; i < indexSz; i++) {
        v[n+i] = suf[0][i << bSzLog];
    }
    build(1, n, n + indexSz, (1 << lg) - n
}

inline void updateBetweenZero(int bid) {
    int bSzLog = (lg+1) >> 1;
    v[n+bid] = suf[0][bid << bSzLog];
    update(1, n, n + indexSz, (1 << lg) -
}

void build(int layer, int lBound, int rBou
```

```
    if (layer >= (int)layers.size()) {
        return;
    }
    int bSz = 1 << ((layers[layer]+1) >> 1
    for (int l = lBound; l < rBound; l +=
        int r = min(l + bSz, rBound);
        buildBlock(layer, l, r);
        build(layer+1, l, r, betweenOffs);
    }
    if (layer == 0) {
        buildBetweenZero();
    } else {
        buildBetween(layer, lBound, rBound
    }
}

void update(int layer, int lBound, int rBo
    if (layer >= (int)layers.size()) {
        return;
    }
    int bSzLog = (layers[layer]+1) >> 1;
    int bSz = 1 << bSzLog;
    int blockIdx = (x - lBound) >> bSzLog;
    int l = lBound + (blockIdx << bSzLog);
    int r = min(l + bSz, rBound);
    buildBlock(layer, l, r);
    if (layer == 0) {
        updateBetweenZero(blockIdx);
    } else {
        buildBetween(layer, lBound, rBound
    }
    update(layer+1, l, r, betweenOffs, x);
}
```

```
    inline SqrtTreeItem query(int l, int r, in
        if (l == r) {
            return v[l];
        }
        if (l + 1 == r) {
            return op(v[l], v[r]);
        }
        int layer = onLayer[clz[(l - base) ^ (
        int bSzLog = (layers[layer]+1) >> 1;
        int bCntLog = layers[layer] >> 1;
        int lBound = (((l - base) >> layers[la
        int lBlock = ((l - lBound) >> bSzLog)
        int rBlock = ((r - lBound) >> bSzLog)
        SqrtTreeItem ans = suf[layer][l];
        if (lBlock <= rBlock) {
            SqrtTreeItem add = (layer == 0) ?
                query(n + lBlock, n + rBlock,
            ) : (
                between[layer-1][betweenOffs +
            );
            ans = op(ans, add);
        }
        ans = op(ans, pref[layer][r]);
        return ans;
    }
public:
    inline SqrtTreeItem query(int l, int r) {
        return query(l, r, 0, 0);
    }

    inline void update(int x, const SqrtTreeIt
        v[x] = item;
```

```
        update(0, 0, n, 0, x);
    }

    SqrtTree(const vector<SqrtTreeItem>& a)
        : n((int)a.size()), lg(log2Up(n)), v(a
        clz[0] = 0;
        for (int i = 1; i < (int)clz.size(); i
            clz[i] = clz[i >> 1] + 1;
        }
        int tlg = lg;
        while (tlg > 1) {
            onLayer[tlg] = (int)layers.size();
            layers.push_back(tlg);
            tlg = (tlg+1) >> 1;
        }
        for (int i = lg-1; i >= 0; i--) {
            onLayer[i] = max(onLayer[i], onLay
        }
        int betweenLayers = max(0, (int)layers
        int bSzLog = (lg+1) >> 1;
        int bSz = 1 << bSzLog;
        indexSz = (n + bSz - 1) >> bSzLog;
        v.resize(n + indexSz);
        pref.assign(layers.size(), vector<Sqrt
        suf.assign(layers.size(), vector<SqrtT
        between.assign(betweenLayers, vector<S
        build(0, 0, n, 0);
    }
};
```

Problems

CodeChef - SEGPROD