ENS Lyon Camp. Day 1. Basic group.
Segment Tree
26 October

Contents

1 Problem statement. Prefix sums. 1

2 Segment tree. Definition 2

3 Segment tree. Construction. 2
3.1 Estimating number of vertex 2
3.2 Algorithm 3

4 Segment tree. Single update 4
4.1 Query for minimum 4
4.2 Update 5

5 Segment tree. Range update. 7

6 Theoretical problems 8

1 Problem statement. Prefix sums.

Consider an array of n elements. The main query is to calculate some statistics of the elements in some range of the array. Of course, each query should be processed effectively. Imagine that there is no request to change the elements in the array.

Prefix sum is an array $psum$ of n elements, that defined in the following way: $psum[k] = \sum_{i=0}^{k} a[i] = psum[k-1] + a[k]$. Time complexity of this solution equals $\mathcal{O}(n)$ for precalculation and $\mathcal{O}(1)$ for a query. Memory complexity is $\mathcal{O}(n)$.

So, the sum of elements in range from l to r could be found as $psum[r] - psum[l-1]$. Of course, this method does not help when function min should be calculated. So the evident question is: what kind of properties should have an operation? It should be associative and have inverse element for each element.

Usually this kind of problem is named RSQ/RMQ (Range Sum Query/Range Minimum (Maximum) Query).

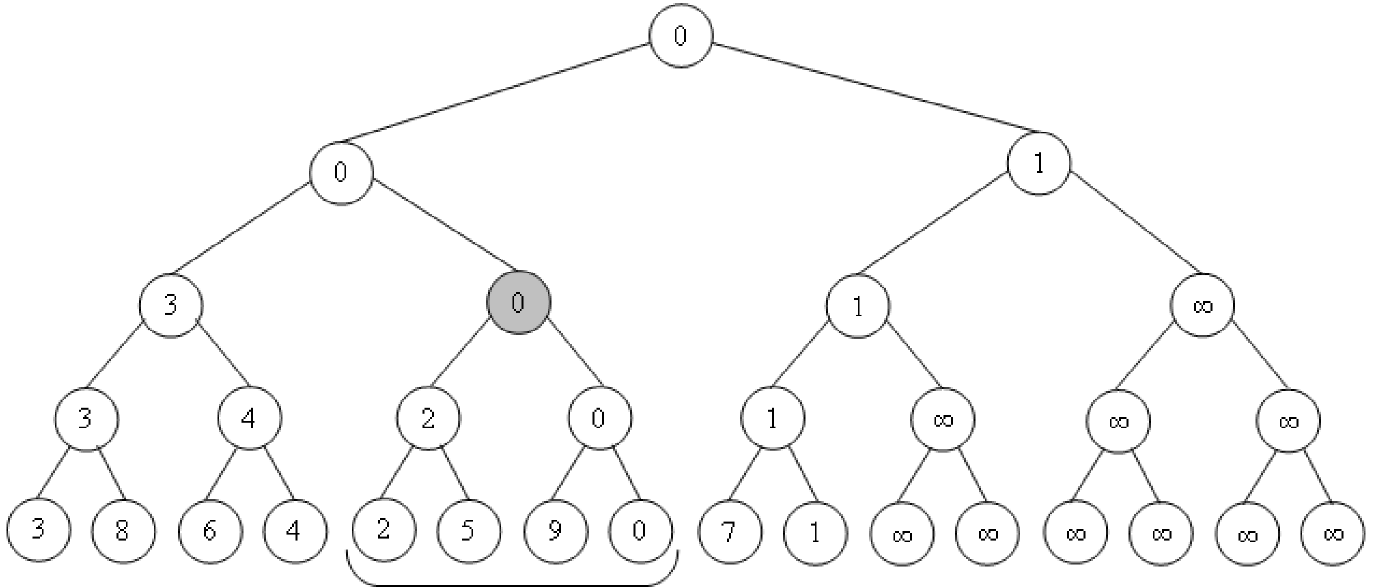

Figure 1: Segment tree for minimum.

2 Segment tree. Definition

Segment tree is a binary tree; each vertex of the segment tree contains a result of operation in the some range. Each vertex, except leaves, has two children and contains the result of the operation on values in its children. Root of the segment tree contains the result of the operation on all elements in the array. If some vertex contains result of the operation in the range, then its left and right children contain a result of operation in the left half and right half of range, respectively. Leaves of the segment tree contain the elements of the given array.

The same question is: what kind of propeties should have an operation? As we see later, the operation should be associative and have the identity element (NB: A set with such operation is named monoid).

3 Segment tree. Construction.

3.1 Estimating number of vertex

To simplify, imagine now that $n = 2^k$. On the bottom layer (leaves) segment tree has n vertices, the next level has $\frac{n}{2}$ vertices, and so on, each level has exactly half of the number of vertices than previous. So, it is possible to estimate total number of vertices in tree as $n+\frac{n}{2}+\frac{n}{4}+\cdots+1 = 2n-1$. If k is the minimal number such that $2^{k-1} < n \leq 2^k$, so $2^k < 2n$ and the total number of vertices in the segment tree is $\leq 2^{k+1} \leq 4n$. That shows that memory is $\mathcal{O}(n)$.

3.2 Algorithm

Again, imagine now that $n = 2^k$ The tree is stored in an array *tree* of size $2n$. Root is stored in $tree[1]$, left and right children of i-th vertex are stored in $tree[2i]$ and $tree[2i + 1]$, respectively. Leaves are stored in elements in the range from n to $2n - 1$. To be specific, we implement the segment tree for minimum. Of course, implementation for other suitable operations is similar.

The main observation is $tree[i] = min(tree[2i], tree[2i+1])$. So the algorithm iterates through vertices from $n - 1$ to 1 and sets value in each vertex as minimum of values in its children.

To make the size of array be equal 2^k, simply add identity elements at the end of the array.

```
// a - initial array of size n
void build()
{
    int N = (1 << (log(n - 1) + 1));
    int[] tree = new int[2 * N];

    // leaves
    for (int i = N; i < 2 * N; i++)
        tree[i] = a[i - N];

    for (int i = N - 1; i > 0; i--)
        tree[i] = min(tree[2 * i], tree[2 * i + 1]);
}
```

It is also possible to build the tree in top-bottom manner. The construction starts from the root. For each vertex, except leaves, algorithm builds left and right subtrees recursively and after that it calculates value in vertex.

```
// a - initial array of size n
// v - the index of current vertex with corresponding range [left, right)
void build(int v, int left, int right)
{
    // leaf case
    if (left == right - 1) {
        tree[v] = a[left];
        return tree[v];
    }

    int med = (left + right) / 2;
    build(2 * v, left, med); // build left subtree
    build(2 * v + 1, med, right); // build right subtree
    tree[v] = min(tree[2 * v], tree[2 * v + 1]);

    return tree[v];
}

// the main call
build(1, 0, n); // array a starts from 0
```

Time complexity of both methods is $\mathcal{O}(n)$.

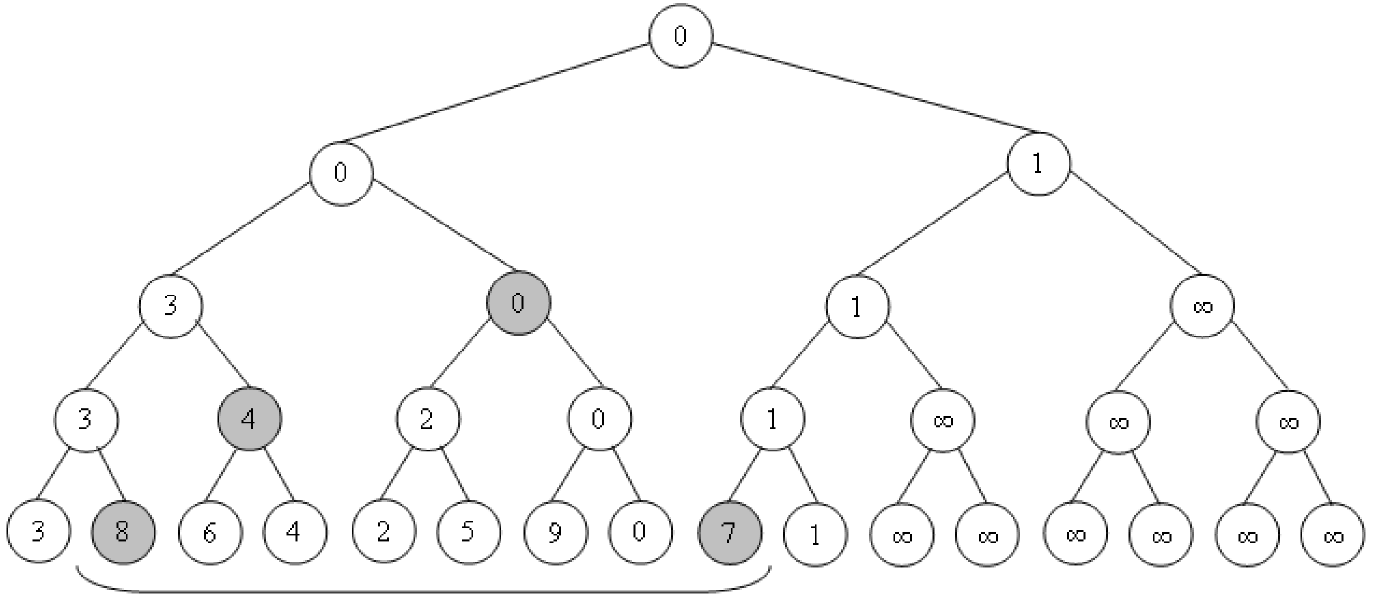

Figure 2: Segment tree. Query.

4 Segment tree. Single update

4.1 Query for minimum

The main idea is to divide given range into disjoint smaller ranges in the following way: every range from partition should be covered by some vertex from tree and total number of ranges is minimal.

Algorithm keeps two pointers l and r. Initially, l and r point to leaves corresponding to the borders of the given range. If l points to the right child of some vertex then this child belongs to answer. Similarly, if r points to the left child of some vertex, this child belongs to answer too. After this check algorithm continues with parents of l and r.

```
// find minimum in range [l, r]
int min_query(int l, int r)
{
    int ans = INF;

    l += N; r += N; // position of borders in tree
    while (l <= r)
    {
        if (l mod 2 == 1) // all right children have odd indexes
            ans = min(ans, T[l]);
        if (r mod 2 == 0) // all left children have even indexes
            ans = min(ans, T[r]);

        // move pointers to the next layer
```

```
        l = (l + 1) / 2, r = (r - 1) / 2;
    }

    return ans;
}
```

It is also possible to process query in top-down manner. Algorithm starts from the root, and for each vertex there are three cases for the mutual arrangement of the range of the vertex and requested range:

- The range of the vertex does not intesect with requested range. In this case algorithm does not include this vertex to the answer.
- The range of the vertex totally lies inside the requested range. In this case algorithm include this vertex into the answer.
- Otherwise, algorithm process query recursively for children of this vertex.

```
// v - the index of current vertex with corresponding range [left, right)
// find minimum in the range [l, r)
int min_query(int v, int left, int right, int l, int r) {
    // range of v totally outside given range [l, r)
    if (right <= l || r <= left) {
        return INF;
    }

    // Range of v totally covered by given range [l, r)
    if (l <= left && right <= r) {
        return tree[v];
    }

    int med = (l + r) / 2;
    int left_min = min_query(2 * v, left, med, l, r); // process left subtree
    int right_min = min_query(2 * v + 1, med, right, l, r); // process right
subtree

    return min (left_min, right_min);
}

// main function
int min_query(int l, int r) {
    return min_query(1, 0, N, l, r);
}
```

Time complexity of both methods is $\mathcal{O}(\log n)$.

4.2 Update

Algorithm goes from updating the leaf to root and updates values in all vertices on the path.

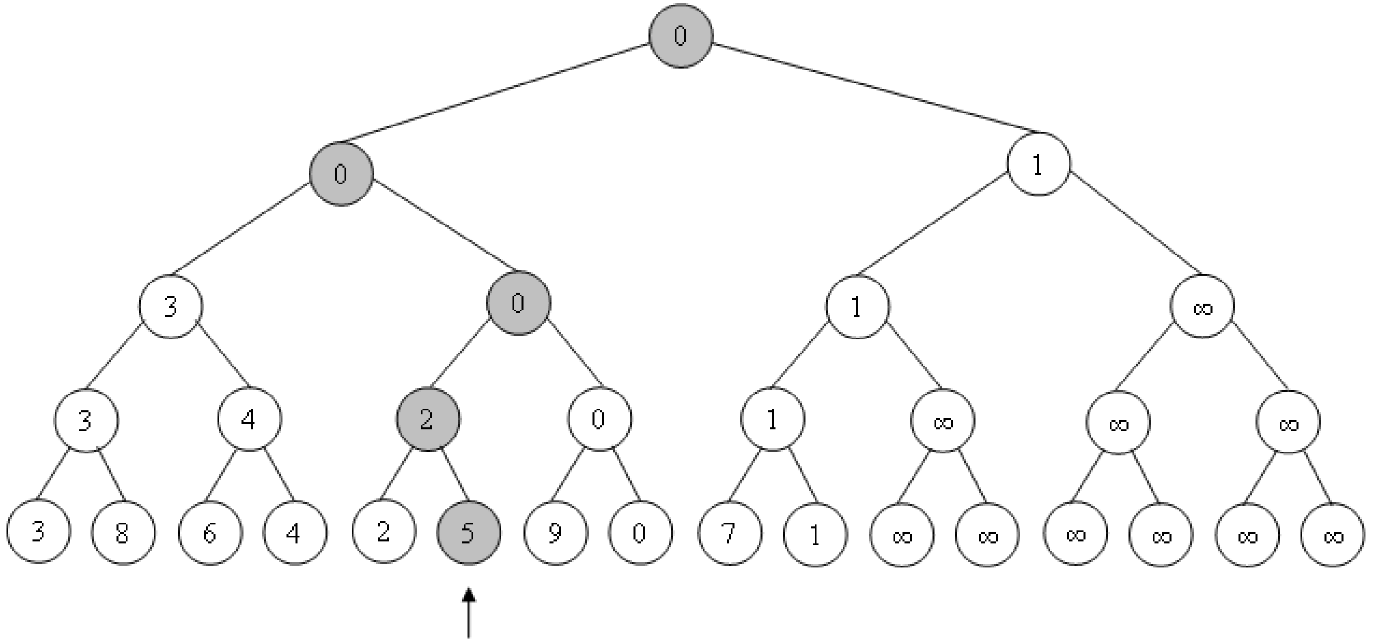

Figure 3: Segment tree. Update.

```
1     // update element at index i with value x
2     void update(int i, int x)
3     {
4         i += N; // position of the element in the tree
5         tree[i] = x;
6
7         // update all segments contained this element
8         while (i > 0) {
9             tree[i] = min(tree[2 * i], tree[2 * i + 1]);
10            i /= 2;
11        }
12    }
```

It is also possible to process query in top-down manner.

```
1     // v - the index of current vertex with corresponding range [left, right)
2     // update element at index i with value x
3     void update(int v, int left, int right, int i, int x)
4     {
5         // element is outside corresponding vertex range
6         if (right <= i || i < left) {
7             return;
8         }
9
10        if (right - left == 1) {
11            tree[v] = x;
12            return;
13        }
```

```
        int med = (l + m) / 2;
        update(2 * v, left, med, i, x); // update left subtree
        update(2 * v + 1, med, right, i, x); // update right subtree

        tree[v] = min(tree[2 * v], tree[2 * v + 1]); // update value in vertex
    }

    // main procedure
    void update(int i, int x) {
        update(1, 0, N, i, x);
    }
```

Time complexity of both methods is $\mathcal{O}(\log n)$.

5 Segment tree. Range update.

Now the problem becomes more complicated. We need to answer on minimum query and add some value on the segment. Now each node keeps not only value in the vertex, but additional field named `inconsistency`. `inconsistency` field, shows how the range of this vertex was updated by the queries. If the node keeps up-to-date value, its inconsistency equals to identity element.

```
    struct Node {
        int value;
        int inconsistency;

        // update value with inconsistency x
        void update(int x) {
            value += x;
            inconsistency += x;
        }
    }

    // Segment tree
    Node tree[];
```

Algorithm maintains the following ivariant: at every time when algorithm visit vertex, this vertex keeps up-to-date value. This invariant is maintained by push procedure. This procedure pushes inconsistency to children for each visit of the vertex. After push the inconsistency of vertex is identety element. Moreover after each visit to vertex, it needs to recalculate value, stored in it.

```
    void push(int v) {
        // update left tree
        if (2 * v < tree.length) {
            tree[2 * v].update(tree[v].inconsistency);
        }

        // update right tree
        if (2 * v + 1 < tree.length) {
            tree[2 * v + 1].update(tree[v].inconsistency);
        }
```

```
        tree[v].inconsistency = 0; // set inconsistency equals identity element
    }
```

Algorithm is similar to the algorithm for simple update. But it is very important to maintain invariant, so algorithm needs to push inconsistency to children before update. **Do not forget to call push function!**

```
    // v - the index of current vertex with corresponding range [left, right)
    // add value x for element in range [l, r)
    void add(int v, int left, int right, int l, int r, int x) {
        if (right <= r || l <= left) {
            return;
        }

        if (l <= left && right <= r) {
            tree[v].update(x);
            return;
        }

        push(v); // push inconsistency to children

        // update childrens
        int med = (l + r) / 2;
        add(2 * v, left, med, l, r, x);
        add(2 * v + 1, med, right, l, r, x);

        tree[v].value = min(tree[2 * v].value, tree[2 * v + 1].value);
    }

    // main procedure
    void add(int l, int r, int x) {
        add(1, 0, N, l, r, x);
    }
```

Time complexity is $\mathcal{O}(\log n)$.

6 Theoretical problems

To better understand this topic, we recommend you to solve this three simple problems.

1. Prove that time complexity for all queries is $O(\log n)$. (Hint: prove the number of ranges in partition do not exceed two. And number of layers is $\log N$).

2. Consider query on the prefix (i.e in range $[0, r)$). What vertices are included into answer?

3. Come up with algorithm that responses on the following queries: finds the k-th non zero element in the array; sets element to non-negative value. What is complexity of your algorithm? (Hint: use segment tree).