Lection 5: Segment Tree and its friends

29.03.2018

Filipp Rukhovich

Hello India x Russia International Bootcamp

RSQ & RMQ: definitions

RSQ (Range Sum Query): given array $= A[0..n-1]$ consisting of n numbers. We are to deal with queries of two types:

1. $assign(index, newValue)$: assign value $newValue$ to $A[index]$;
2. $findSum(l, r)$: find value $A[l] + A[l+1] + \ldots + A[r]$ on subsegment $A[l..r]$.

RMQ (Range Min Query): given array $= A[0..n-1]$ consisting of n numbers. We are to deal with queries of two types:

1. $assign(index, newValue)$: assign value $newValue$ to $A[index]$;
2. $findMin(l, r)$: find value $min(A[l], A[l+1], \ldots, A[r])$ on subsegment $A[l..r]$.

Static RSQ: partial sums

Firstly, consider a problem of "static RSQ", in which we don't need to perform assignments and change an array.

To do it effectively, build an array of partial sums:

$partialSums[i] := A[0] + A[1] + \ldots A[i], 0 \leqslant i < n.$

Such sums can be precalculated in $O(n)$ using a property:

$partialSums[i+1] = partialSums[i] + A[i+1].$

Then, sum on subsegment $[l, r]$ can be found in $O(1)$ as $partialSums[r] - partialSums[l-1]$ (or as $partialSums[r]$ in case $l = 0$).

This solution is optimal asymptotically.

Static RMQ: sparse table

Unfortunately, *min* is not inversable operation, as *sum*; it means an impossibility to implement an analogue of partial sums for static RMQ.

But we can use an *idempotency* of *min*, that is a fact that $min(a, a) = a$.

Calculate for an array $A[0..n-1]$ *Sparse Table* $sp[0..n-1][0..\lceil log_2 n \rceil]$, such that

$sp[i][j] = min\{a[k] | i \leqslant k < min(i + 2^j, n)\}$.

It can be done in $O(n \ log \ n)$, using the facts that:

- $sp[i][0] = a[i]$;
- $sp[i][j+1] = min(sp[i][j], sp[min(i + 2^j, n-1)][j])$.

Static RMQ: sparse table

Having such table, one can calculate minimum on any subsegment $A[l, r]$ with complexity $O(1)$!

Details of the algorithm:

- Let $lv = \lfloor log_2(r - l + 1) \rfloor$; in other words, lv is maximal integer number such that $2^{lv} \leqslant r - l + 1$.
- Then $min(A[l, r]) = min(minl, minr)$; in this formula:
  - $minl := min(A[l, l + 2^{lv} - 1]) = sp[l][lv]$;
  - $minr = min(A[r - 2^{lv} + 1, r]) = sp[r - 2^{lv} + 1][lv]$.

So, the only nontrivial action we should do is to calculate a logarithm.

In practice, values $lv(x)$ are calculated in precalculation for all $x = 1, 2, \ldots, n$.

But what should one do if the array can be changed?

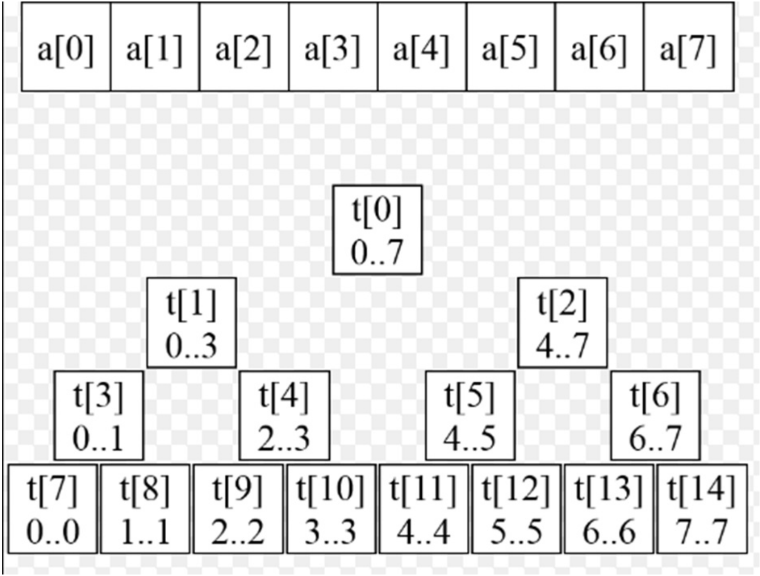
Segment Tree: description - 1

To perform queries of both types successfully, define the following data structure (we'll work with RSQ problem; for RMQ, the structure is the same):

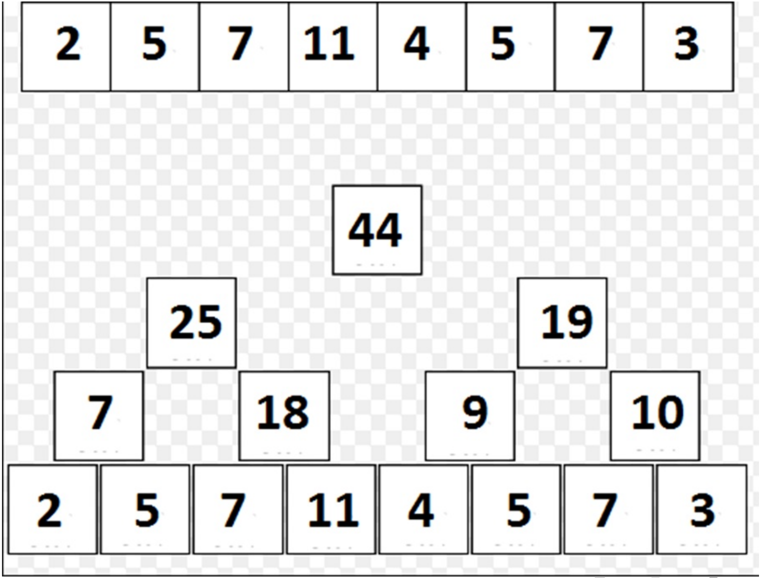
- Suppose for convenience that $n = 2^k$ for some integer k;
- As in sparse table, choose some subsegments and store and support sum of elements in them.
- In this structure, that will be following segments:
  - $[0, 0], [1, 1], \ldots, [n-1, n-1]$ (n subsegments);
  - $[0, 1], [2, 3], \ldots, [n-2, n-1]$ ($n/2$ subsegments);
  - $[0, 3], [4, 7], \ldots, [n-4, n-1]$ ($n/4$ subsegments);
  - $\ldots$;
  - $[0..n-1]$ (1 subsegment).

Totally, we store and support sums on exactly $2n - 1$ subsegments.

Segment Tree: description - 2



Segment Tree: description - 3



Segment Tree: representation

As one can see for the picture, sums for all interesting subsegments are stored in an array $t[0..2n-2]$ in such a way that:

- vertex, or element of array, 0 is corresponding with subsegment $[0..n-1]$;
- vertices $n-1, n, \ldots, 2n-2$, are corresponding with subsegments $[0..0]$, $[1..1]$, …, $[2n-2..2n-2]$;
- any vertex with number $v < n-1$ has two children $2v+1$ and $2v+2$, s.t. if vertex v is corresponding with subsegment $[l..r]$, then children are corresponding with subsegments $[l..mid]$ and $[mid+1..r]$; here $mid := \lfloor (l+r)/2 \rfloor$.
- any vertex with number $v > 0$ has a parent $\lfloor (v-1)/2 \rfloor$.

But how to process the queries?

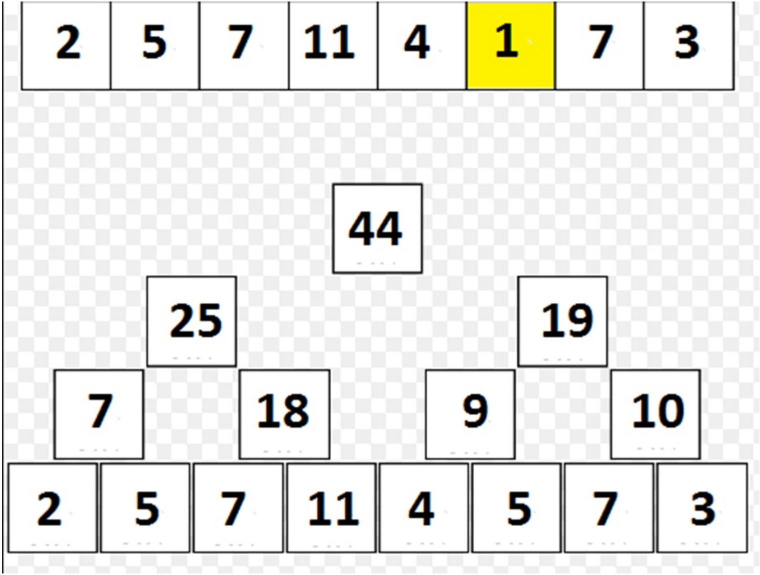
Segment Tree: assignment query - 1

Suppose one has an $A[num]$ changing query.

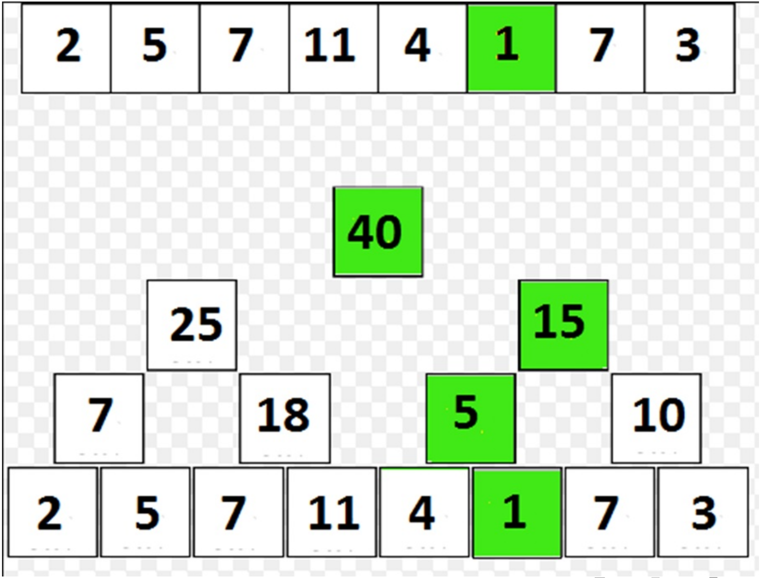
Then, only values in vertex $num + n - 1$ and in all of its ancestor are changed.

There are exactly $\lfloor log_2 n \rfloor + 1$ such vertices; so the tree can be updated in $O(log\ n)$ time.

Segment Tree: assignment query - 2



Segment Tree: assignment query - 3



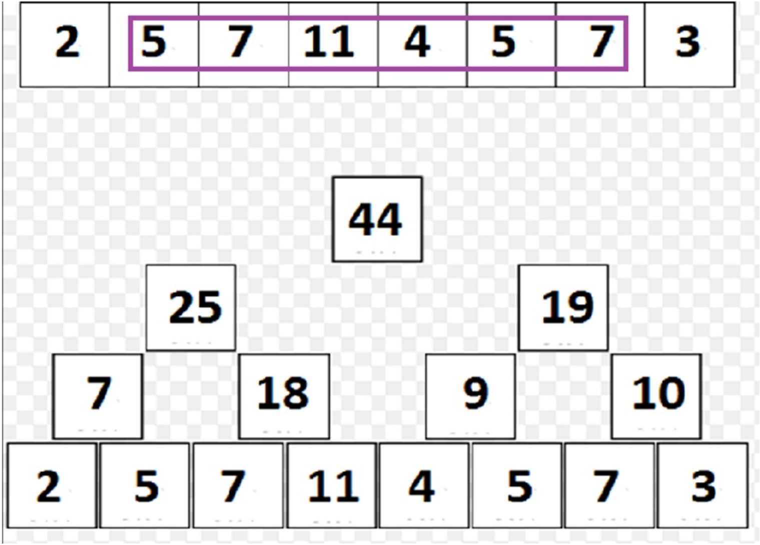
Segment Tree: sum query: "from below" way - 1

We'll consider two different ways to find a sum on subseqment $[l, r]$, $0 \leqslant l \leqslant r < n$.

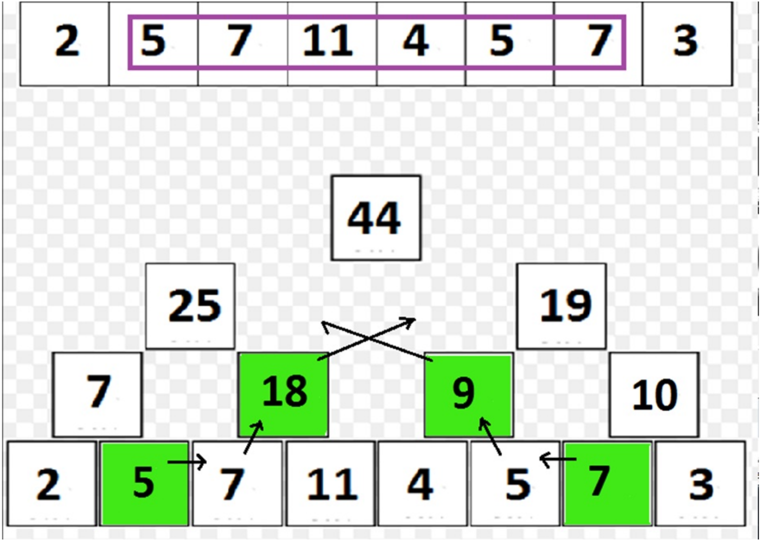
Way 1 - "from below", or "non-recursive":

- Let keep current result in variable *ans*; at the beginning, $ans = 0$.
- Notice that all elements of subsegment $[l..r]$ except possibly ends are "covered" by subsegments of length 2 which are corresponding with some vertices of the tree and also are subsegments of subsegment $[l..r]$;
- Moreover, l-th element is covered by such element iff $l\%2 == 1$, and r-th - iff $r\%2 == 0$.
- Add if one needs l-th and/or r-th elements in *ans*, go one level higher; one ths level, repeat the procedure.
- On each level, there are only $O(1)$ actions are performed; so, sum can be found by our algorithm on $O(log_2 n)$ of operations!

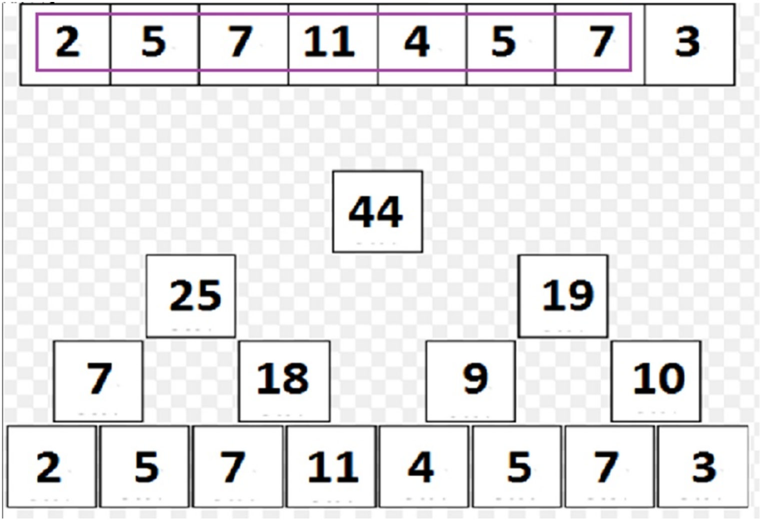
Segment Tree: sum query: "from below" way - 2



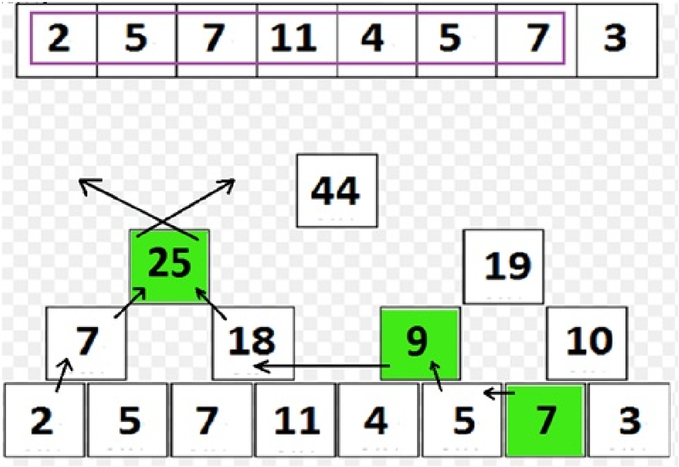
Segment Tree: sum query: "from below" way - 3



Segment Tree: sum query: "from below" way - 4



Segment Tree: sum query: "from below" way - 5



Segment Tree: sum query: "from below" way - 6

```
uint32 n; // size of array a
uint32 shift; // start of 1-length segment vertices in tree
std::vector<int> a(n), tree;

int getParent(int v) {
    return (v - 1) / 2;
}

int recalculate(int v) {
    tree[v] = tree[2*v+1] + tree[2*v+2];
}

uint32 calculateShift(uint32 n) {
    uint32 shift = 1;
    while (n > shift)
        shift += shift;
    --shift;
    return shift;
}

void buildTree() {
    shift = calculateShift(n);
    tree.resize(2*shift + 1);
    std::copy(a.begin(), a.end(), tree.begin() + shift);
    for (int i = static_cast<int>(shift) - 2; i >= 0; --i)
        recalculate(i);
}
```

Segment Tree: sum query: "from below" way - 7

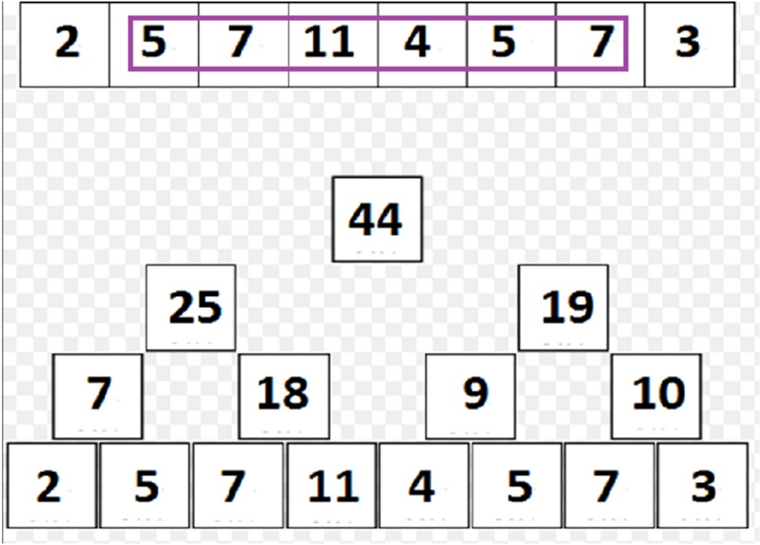
```
void assign(uint32 index, int x) {
    a[index] = x;
    int v = index + step - 1;
    tree[v] = x;
    for (v = getParent(v); v >= 0; v = getParent(v)
        recalculate(v);
}

int findSum(uint32 l, uint32 r) {
    int ans = 0;
    l += shift;
    r += shift;
    while (l <= r) {
        if (!(l&1))
            ans += tree[l++];
        if (r&1)
            ans += tree[r--];
        l = getParent(l);
        r = getParent(r);
    }
    return ans;
}
```

Segment Tree: sum query: "from above" way - 1

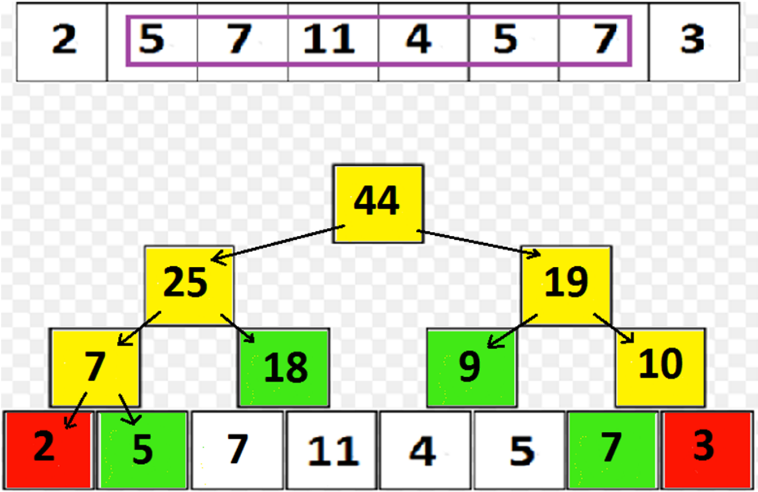
Then, consider a "recursive" way to find sum on subsegment $[ql..qr]$.

- We will start from root and work, if need, recursively with children;
- Suppose that at a current moment of time we are in vertex v with corresponding subsegment $[l..r]$.
- If subsegments $[l..r]$ and $[ql..qr]$ are non-intersecting, then return 0;
- If $[l..r] \subseteq [ql..qr]$, then return $tree[v]$;
- Otherwise, work with children recursively and return sum of results of launches.

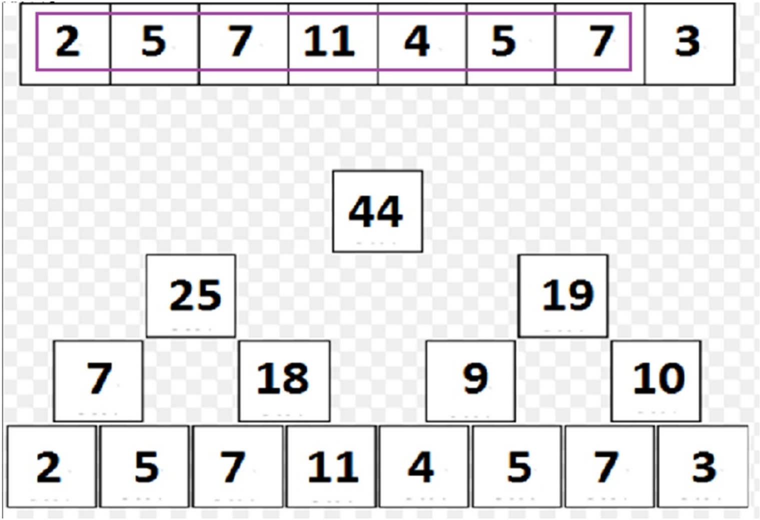
Segment Tree: sum query: "from above" way - 2



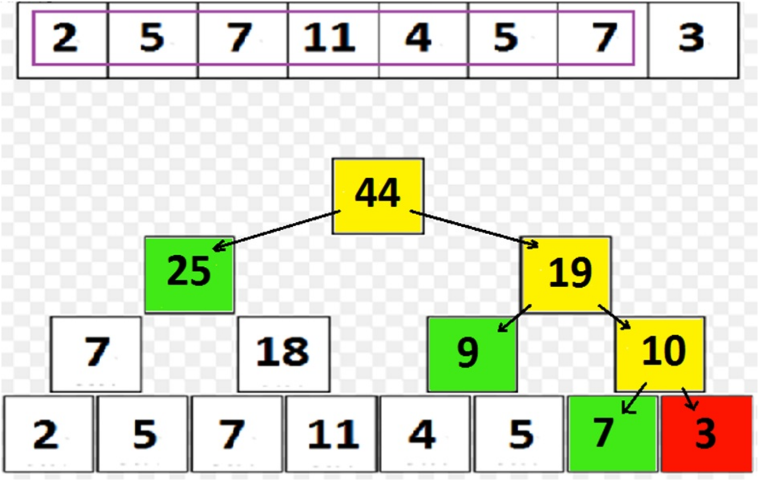
Segment Tree: sum query: "from above" way - 3



Segment Tree: sum query: "from above" way - 4



Segment Tree: sum query: "from above" way - 5



Segment Tree: sum query: "from above" way - 6

```
void assign(uint32 v, uint32 l, uint32 r, uint32 index, int x) {
    if (r < x || l > x)
        return;
    if (l == r) {
        tree[v] = x;
        return;
    }

    uint32 mid = (l + r) / 2;
    assign(2*v+1, l, mid, ql, qr);
    assign(2*v+2, mid+1, r, ql, qr);
    recalculate(v);
}

int findSum(uint32 v, uint32 l, uint32 r, uint32 ql, uint32 qr) {
    if (ql <= l && l <= r && r <= qr)
        return tree[v];
    if (qr < l || r < ql)
        return 0;
    uint32 mid = (l + r) / 2;
    return findSum(2*v+1, l, mid, ql, qr) + findSum(2*v+2, mid+1, r, ql, qr);
}
```

Segment Tree: assignment on subsegment - 1

Now, suppose that besides first two operations, one is to perform queries of new, third, type: given number l, r, x, one should assign number x to elements $A[l], A[l+1], \ldots, A[r]$ of array A.

To support segment tree in current way, it's necessary to change a linear number of vertices, which makes the structure meaningless.

The following method gives us our $O(log\ n)$ efficiency:

- All operations should be performed "from above";
- At each vertex v, store bool additional variables: bool *isAssigned* and int *valueAssigned*.
- At any vertex v, if $isAssigned = false$, then *valueAssigned* is meaningless;
- Otherwise, the meaning of *valueAssigned* is "all elements of subsegment corresponding to v are equal to *valueAssigned*, but in fact, descendants of v don't know about it".

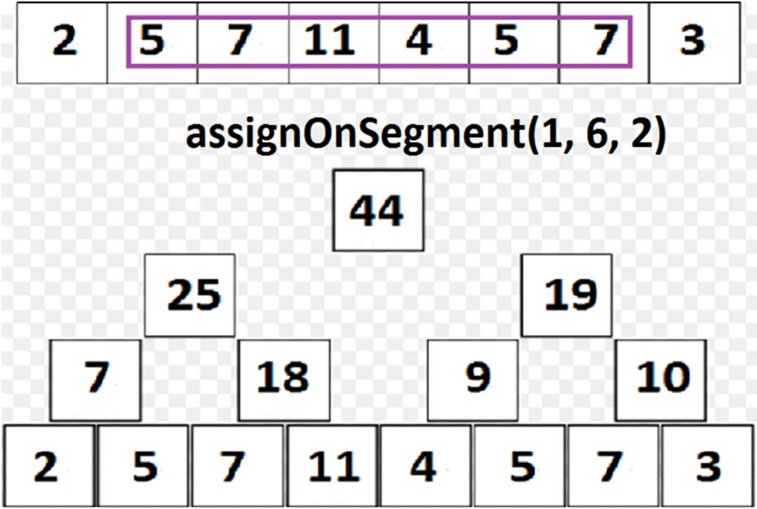
Segment Tree: assignment on subsegment - 2

- function *assignOnSegment* is implemented as *findSum*; only "green" (on the following slides) vertex should learn information about new assignment.
- If during some operation, one should go to children of v, and $tree[v].isAssigned = true$, then before recursive launches, we are to "inform" children about *valueAssigned*, and then assign *false* to $tree[v].isAssigned$.

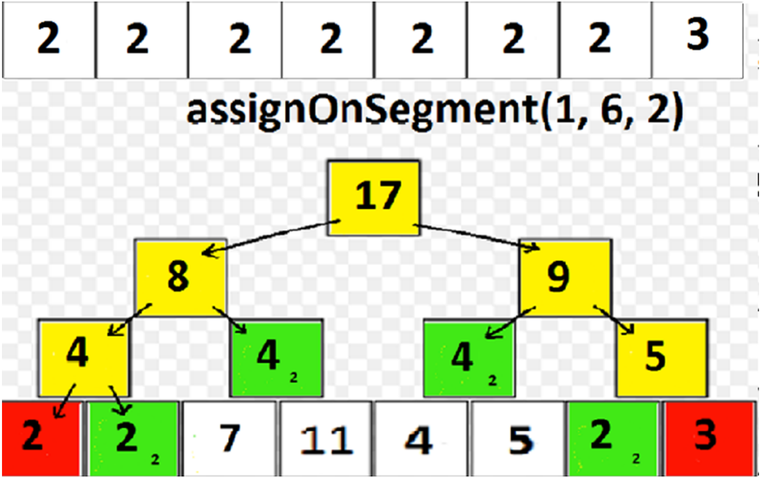
This method is called "method of lazy propagation".

The main principle of the method is "if we work with vertex v, the it knows about all assignments it should know".

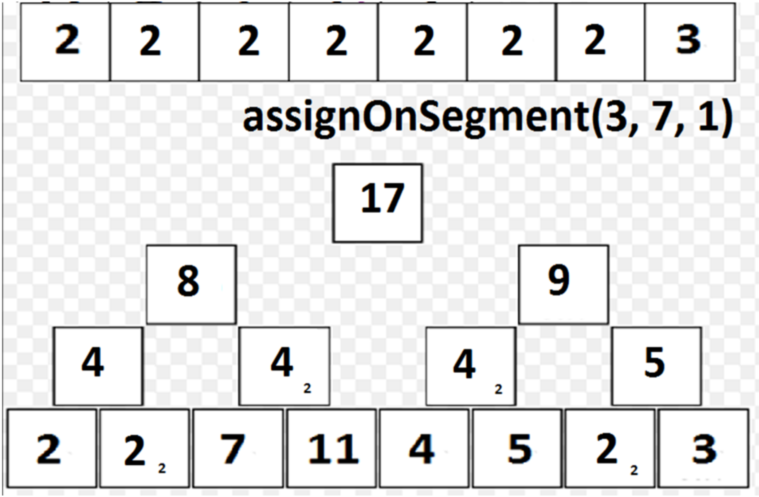
Segment Tree: assignment on subsegment - 3



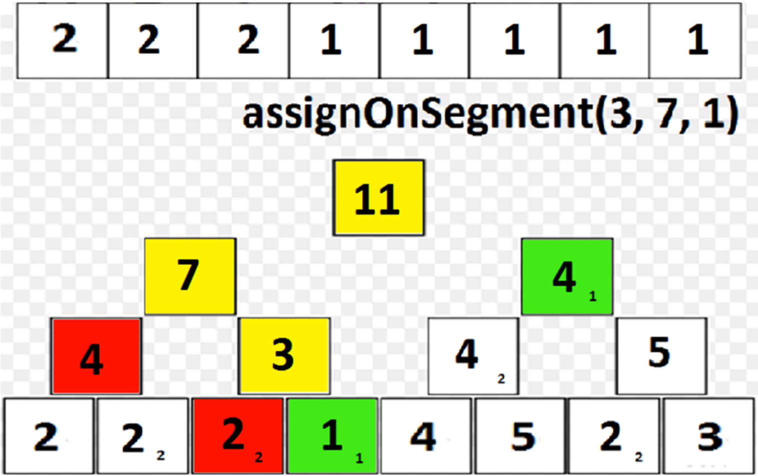
Segment Tree: assignment on subsegment - 4



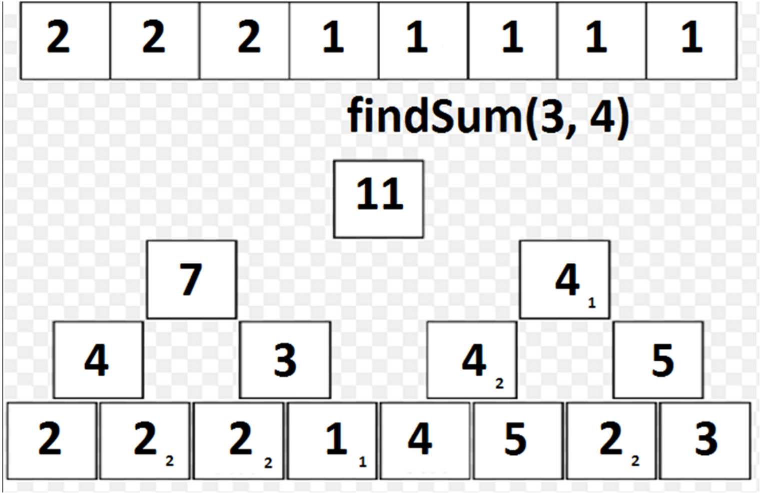
Segment Tree: assignment on subsegment - 5



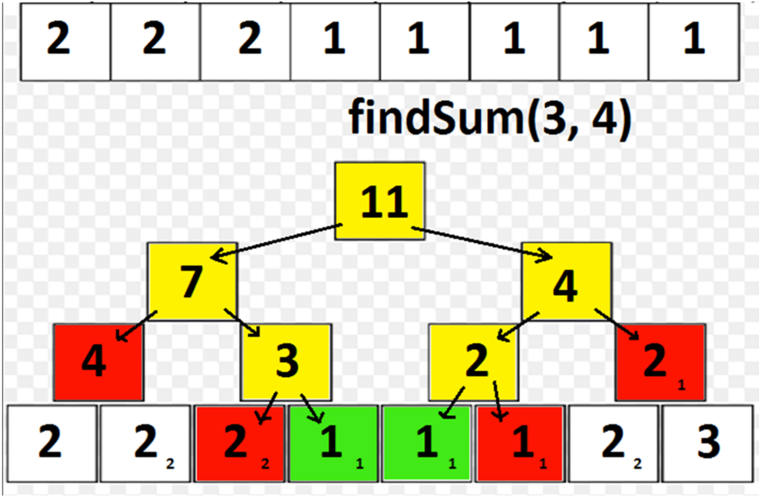
Segment Tree: assignment on subsegment - 6



Segment Tree: assignment on subsegment - 7



Segment Tree: assignment on subsegment - 8



Segment Tree: assignment on subsegment - 9

```
struct Node {
    int sum, assignedValue;
    bool isAssigned;
};

uint32 n; // size of array a
uint32 shift; // start of 1-length segment vertices in tree
std::vector<int> a(n);
std::vector<Node> tree;

int getParent(int v) {
    return (v - 1) / 2;
}

int recalculate(int v) {
    tree[v].sum = tree[2*v+1].sum + tree[2*v+2].sum;
}

uint32 calculateShift(uint32 n) {
    uint32 shift = 1;
    while (n > shift)
        shift += shift;
    --shift;
    return shift;
}
```

Segment Tree: assignment on subsegment - 10

```
void buildTree() {
    shift = calculateShift(n);
    tree.resize(2*shift + 1);
    for (int i = 0; i < n; ++i)
        Node[shift + i] = Node(a[i], 0, false);

    for (int i = static_cast<int>(shift) - 2; i >= 0; --i) {
        recalculate(i);
        tree[i].isAssigned = false;
    }
}

void assignOnSubtree(uint32 v, uint32 l, uint32 r, int x) {
    tree[v].isAssigned = true;
    tree[v].assignedValue = x;
    tree[v].sum = x * (r - l + 1);
}

void pushDown(uint32 v, uint32 l, uint32 r) {
    if (!tree[v].isAssigned || v >= shift)
        return;
    int mid = (l + r)/2;
    assignOnSubtree(v*2+1, l, mid, tree[v].valueAssigned);
    assignOnSubtree(v*2+2, mid+1, r, tree[v].valueAssigned);
}
```

Segment Tree: assignment on subsegment - 11

```
void assignOnSegment(uint32 v, uint32 l, uint32 r, uint32 ql, uint32 qr, int x) {
    if (r < ql || l > qr)
        return;
    if (ql <= l && r <= qr) {
        assignOnSubtree(v, l, r, x);
        return;
    }

    pushDown(v, l, r);
    uint32 mid = (l + r) / 2;
    assignOnSegment(2*v+1, l, mid, ql, qr);
    assignOnSegment(2*v+2, mid+1, r, ql, qr);
    recalculate(v);
}

int findSum(uint32 v, uint32 l, uint32 r, uint32 ql, uint32 qr) {
    if (ql <= l && l <= r && r <= qr)
        return tree[v];
    if (qr < l || r < ql)
        return 0;

    pushDown(v, l, r);
    uint32 mid = (l + r) / 2;
    return findSum(2*v+1, l, mid, ql, qr) + findSum(2*v+2, mid+1, r, ql, qr);
}
```

Scanning line: example problem

Segment trees are very helpful in many problems which can be solved by method of scanning line.

To see what is the method, try to solve the following problem:

- Given $n \leqslant 10^5$ rectangles on the plane; sides of each of them are parallel to axis; coordinates of angles are also integer and are from 0 to $2 * 10^5$.
- Find a cell which is covered by as maximal number of rectangles as possible.

Scanning line: solution of example problem - 1

To solve the problem, we imagine a vertical line which moves from left to right. At each moment of time, the line intersects a column of cells; we build an array $A[1..2 * 10^5]$ and support the following invariant:

- At each moment of time, $A[i]$ should be equal to a number of rectangles which are covering the cell in i-th row and column scanning line goes through at this moment of time.

At the beginning, all elements of A are equal to zero.

But then, when scanning line moves through rectangles, A will be changed. All changes occurs at the moments of time when does some rectangular begin and/or end; the change is adding (for beginning) or subtracting of 1 from elements of some subsegment of A; this can be done effectively in $O(log\ n)$ using segment tree.

Scanning line: solution of example problem - 2

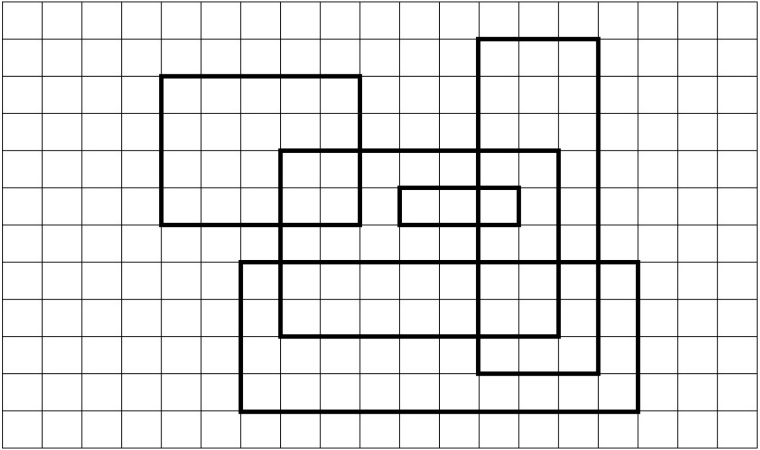
To emulate the process, store $2n$ *events* to a vector. Here, each *event* is a class containing information about beginning or ending of some rectangle: type (beginning or ending), time (x-coordinate) and segment of y-coordinates of cells covered by rectangle.

Sort these *events* by x; after that, go through all *events* and add/subtract 1 on subsegment corresponding to current event.

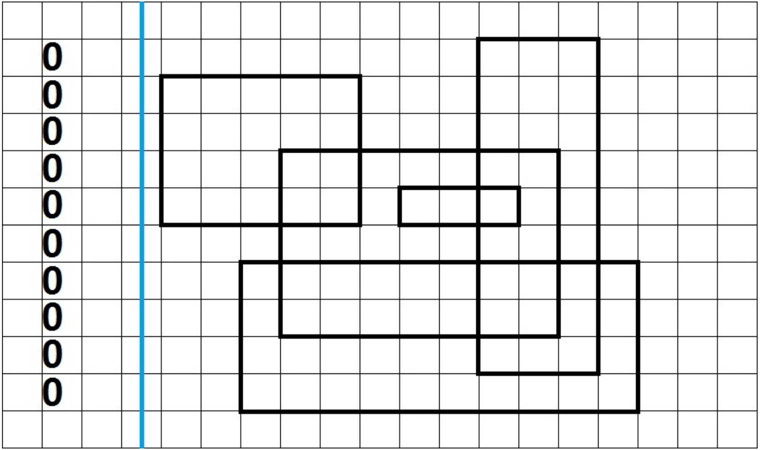
To find an answer, one can just find maximum element between each two neighbouring events with different x-s (it can be done in $O(1)$ using the same segment tree).

The algorithm works in $O(n\ log\ n)$ time.

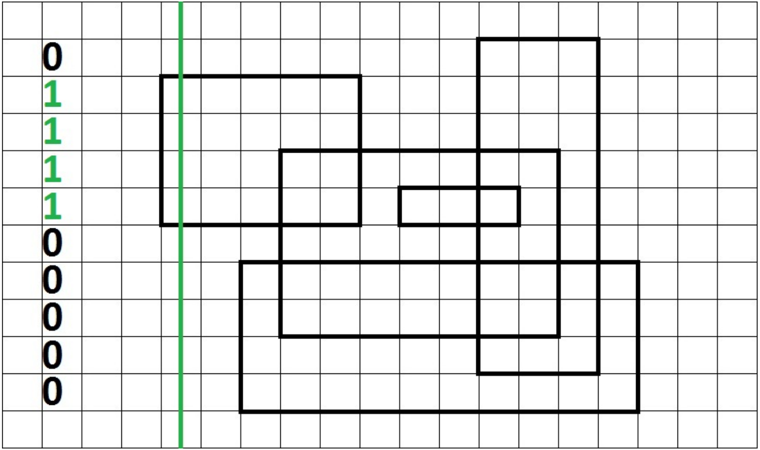
Scanning line: solution of example problem - 3



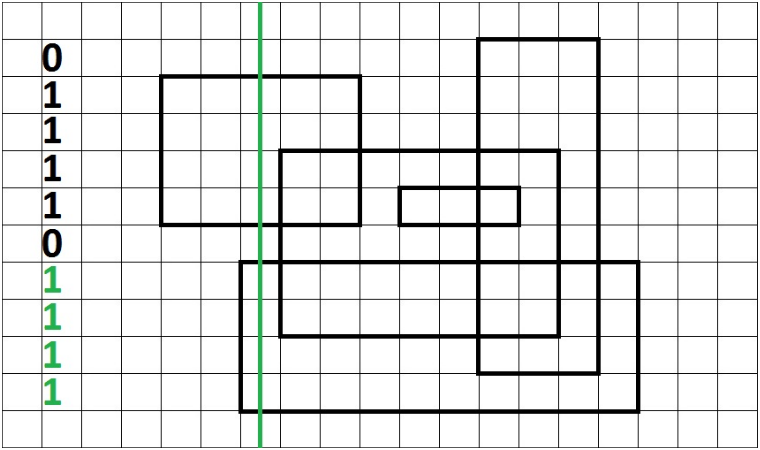
Scanning line: solution of example problem - 4



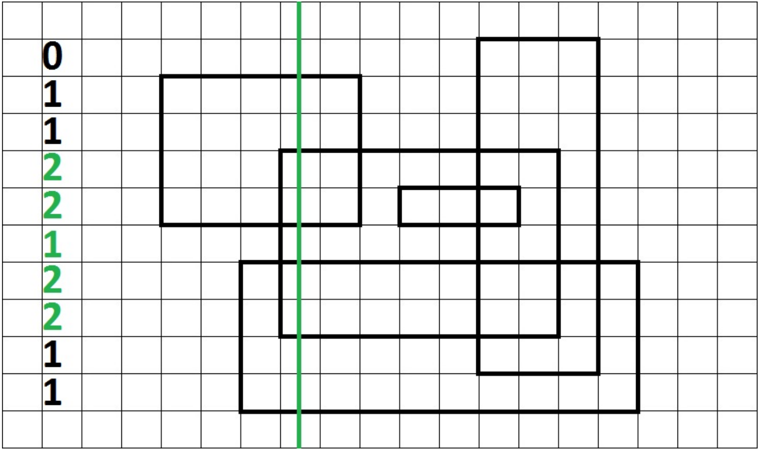
Scanning line: solution of example problem - 5



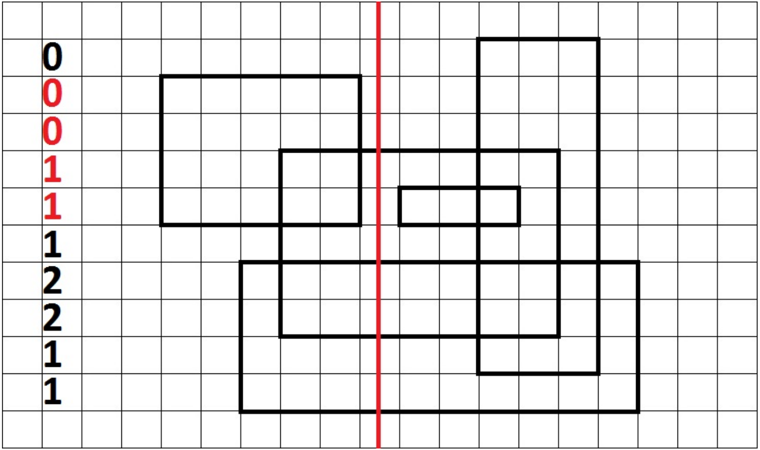
Scanning line: solution of example problem - 6



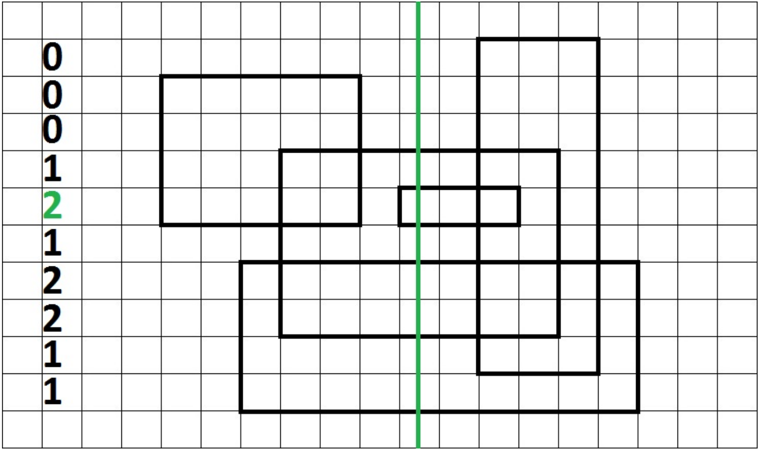
Scanning line: solution of example problem - 7



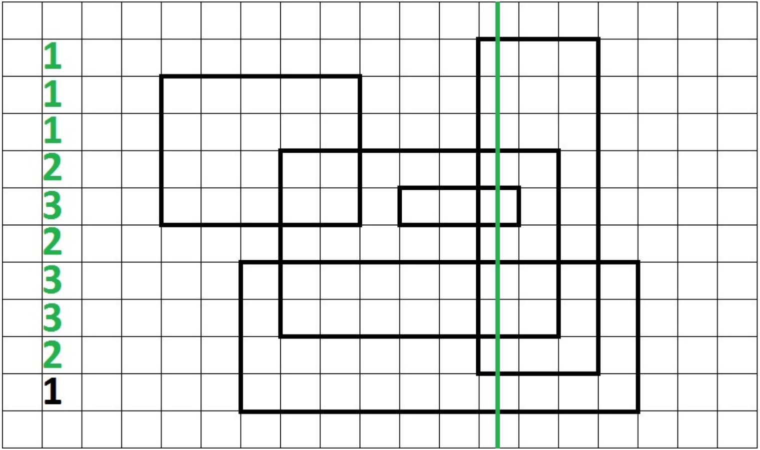
Scanning line: solution of example problem - 8



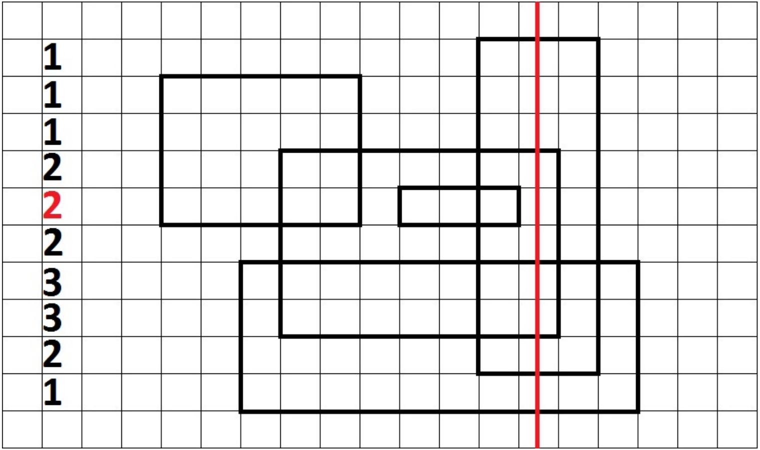
Scanning line: solution of example problem - 9



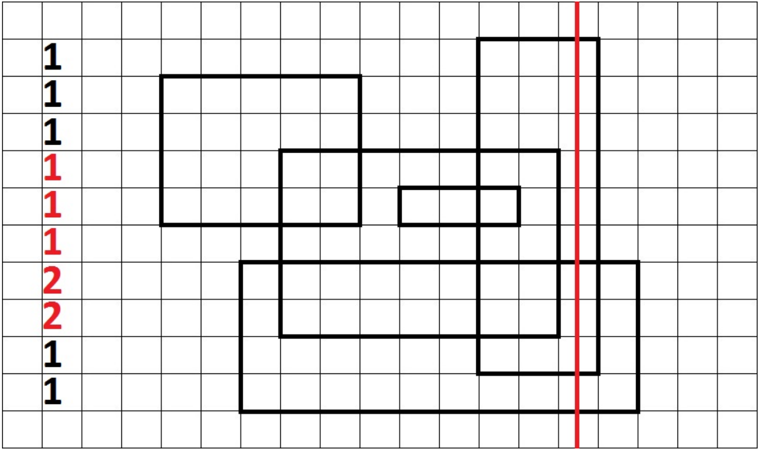
Scanning line: solution of example problem - 10



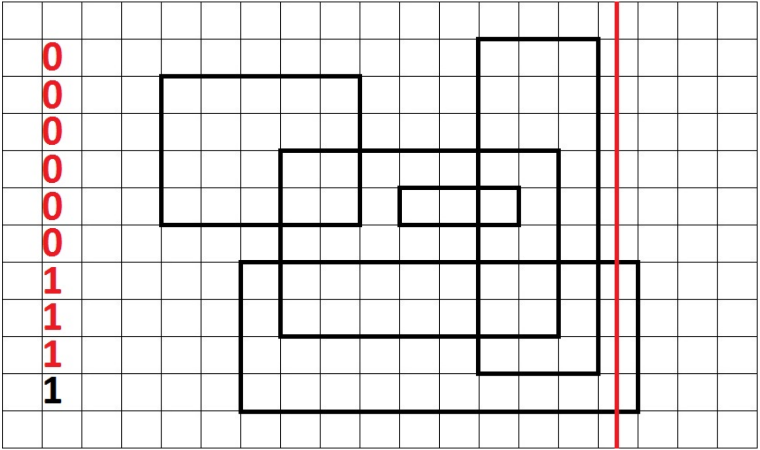
Scanning line: solution of example problem - 11



Scanning line: solution of example problem - 12



Scanning line: solution of example problem - 13



Scanning line: solution of example problem - 14