

[HOME](#) [TOP](#) [CATALOG](#) [CONTESTS](#) [GYM](#) [PROBLEMSET](#) [GROUPS](#) [RATING](#) [EDU](#) [API](#) [CALEN](#)[AL.CASH](#) [BLOG](#) [TEAMS](#) [SUBMISSIONS](#) [GROUPS](#) [CONTESTS](#) [PROBLEMSETTING](#)

Al.Cash's blog

Efficient and easy segment trees

By **Al.Cash**, 10 years ago, 

This is my first attempt at writing something useful, so your suggestions are welcome.

Most participants of programming contests are familiar with segment trees to some degree, especially having read this articles <http://codeforces.com/blog/entry/15890>, http://e-maxx.ru/algo/segment_tree (Russian only). If you're not — don't go there yet. I advise to read them after this article for the sake of examples, and to compare implementations and choose the one you like more (will be kinda obvious).

Segment tree with single element modifications

Let's start with a brief explanation of segment trees. They are used when we have an array, perform some changes and queries on continuous segments. In the first example we'll consider 2 operations:

1. modify one element in the array;
2. find the sum of elements on some segment. .

Perfect binary tree

I like to visualize a segment tree in the following way: [image link](#)

1: [0, 16)															
2: [0, 8)								3: [8, 16)							
4: [0, 4)				5: [4, 8)				6: [8, 12)				7: [12, 16)			
8: [0, 2)	9: [2, 4)	10: [4, 6)		11: [6, 8)		12: [8, 10)		13: [10, 12)		14: [12, 14)		15: [14, 16)			
16: 0	17: 1	18: 2	19: 3	20: 4	21: 5	22: 6	23: 7	24: 8	25: 9	26: 10	27: 11	28: 12	29: 13	30: 14	31: 15

Notation is *node_index: corresponding segment* (left border included, right excluded). At the bottom row we have our array (0-indexed), the leaves of the tree. For now suppose it's length is a power of 2 (16 in the example), so we get perfect binary tree. When going up the tree we take pairs of nodes with indices $(2 * i, 2 * i + 1)$ and combine their values in their parent with index i . This way when we're asked to find a sum on interval $[3, 11)$, we need to sum up only values in the nodes 19, 5, 12 and 26 (marked with bold), not all 8 values inside the interval. Let's jump directly to implementation (in C++) to see how it works:

```
const int N = 1e5; // limit for array size
int n; // array size
int t[2 * N];

void build() { // build the tree
    for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void modify(int p, int value) { // set value at position p
    for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}

int query(int l, int r) { // sum on interval [l, r)
    int res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l&1) res += t[l++];
        if (r&1) res += t[--r];
    }
    return res;
}

int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) scanf("%d", t + n + i);
}
```

```

    build();
    modify(0, 1);
    printf("%d\n", query(3, 11));
    return 0;
}

```

That's it! Fully operational example. Forget about those cumbersome recursive functions with 5 arguments!

Now let's see why this works, and works very efficient.

1. As you could notice from the picture, leaves are stored in continuous nodes with indices starting with n , element with index i corresponds to a node with index $i + n$. So we can read initial values directly into the tree where they belong.
2. Before doing any queries we need to build the tree, which is quite straightforward and takes $O(n)$ time. Since parent always has index less than its children, we just process all the internal nodes in decreasing order. In case you're confused by bit operations, the code in *build()* is equivalent to `t[i] = t[2*i] + t[2*i+1]`.
3. Modifying an element is also quite straightforward and takes time proportional to the height of the tree, which is $O(\log(n))$. We only need to update values in the parents of given node. So we just go up the tree knowing that parent of node p is $p / 2$ or `p >> 1`, which means the same. `p ^ 1` turns $2 * i$ into $2 * i + 1$ and vice versa, so it represents the second child of p 's parent.
4. Finding the sum also works in $O(\log(n))$ time. To better understand it's logic you can go through example with interval $[3, 11)$ and verify that result is composed exactly of values in nodes 19, 26, 12 and 5 (in that order).

General idea is the following. If l , the left interval border, is odd (which is equivalent to `l & 1`) then l is the right child of its parent. Then our interval includes node l but doesn't include it's parent. So we add `t[l]` and move to the right of l 's parent by setting $l = (l + 1) / 2$. If l is even, it is the left child, and the interval includes its parent as well (unless the right border interferes), so we just move to it by setting $l = l / 2$. Similar argumentation is applied to the right border. We stop once borders meet.

No recursion and no additional computations like finding the middle of the interval are involved, we just go through all the nodes we need, so this is very efficient.

Arbitrary sized array

For now we talked only about an array with size equal to some power of 2, so the binary tree was perfect. The next fact may be stunning, so prepare yourself.

The code above works for any size n .

Explanation is much more complex than before, so let's focus first on the advantages it gives us.

1. Segment tree uses exactly $2 * n$ memory, not $4 * n$ like some other implementations offer.
2. Array elements are stored in continuous manner starting with index n .
3. All operations are very efficient and easy to write.

You can skip the next section and just test the code to check that it's correct. But for those interested in some kind of explanation, here's how the tree for $n = 13$ looks like: [image link](#)

1: --															
2: [3, 11)								3: --							
4: [3, 7)				5: [7, 11)				6: --				7: [1, 3)			
8: [3, 5)		9: [5, 7)		10: [7, 9)		11: [9, 11)		12: [11, 13)		13: 0		14: 1		15: 2	
16: 3	17: 4	18: 5	19: 6	20: 7	21: 8	22: 9	23: 10	24: 11	25: 12						

It's not actually a single tree any more, but a set of perfect binary trees: with root 2 and height 4, root 7 and height 2, root 12 and height 2, root 13 and height 1. Nodes denoted by dashes aren't ever used in *query* operations, so it doesn't matter what's stored there. Leaves seem to appear on different heights, but that can be fixed by cutting the tree before the node 13 and moving its right part to the left. I believe the resulting structure can be shown to be isomorphic to a part of larger perfect binary tree with respect to operations we perform, and this is why we get correct results.

I won't bother with formal proof here, let's just go through the example with interval $[0, 7)$. We have $l = 13, r = 20$, `l&1 => add t[13]` and borders change to $l = 7, r = 10$. Again `l&1 => add t[7]`, borders change to $l = 4, r = 5$, and suddenly nodes are at the same height. Now we have `r&1 => add t[4 = --r]`, borders change to $l = 2, r = 2$, so we're finished.

Modification on interval, single element access

Some people begin to struggle and invent something too complex when the operations are inverted, for example:

1. add a value to all elements in some interval;
2. compute an element at some position.

But all we need to do in this case is to switch the code in methods *modify* and *query* as follows:

```
void modify(int l, int r, int value) {
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l&1) t[l++] += value;
        if (r&1) t[--r] += value;
    }
}


int query(int p) {
    int res = 0;
    for (p += n; p > 0; p >>= 1) res += t[p];
    return res;
}
```

If at some point after modifications we need to inspect all the elements in the array, we can push all the modifications to the leaves using the following code. After that we can just traverse elements starting with index n . This way we reduce the complexity from $O(n \log(n))$ to $O(n)$ similarly to using *build* instead of n modifications.

```
void push() {
    for (int i = 1; i < n; ++i) {
        t[i<<1] += t[i];
        t[i<<1|1] += t[i];
        t[i] = 0;
    }
}
```

Note, however, that code above works only in case the order of modifications on a single element doesn't affect the result. Assignment, for example, doesn't satisfy this condition. Refer to section about lazy propagation for more information.

Non-commutative combiner functions

For now we considered only the simplest combiner function — addition. It is commutative, which means the order of operands doesn't matter, we have $a + b = b + a$. The same applies to *min* and *max*, so we can just change all occurrences of  to one of those functions and be fine. But don't forget to initialize query result to infinity instead of 0.

However, there are cases when the combiner isn't commutative, for example, in the problem [380C - Сereja и скобочки](http://codeforces.com/blog/entry/10363), tutorial available here <http://codeforces.com/blog/entry/10363>. Fortunately, our implementation can easily support that. We define structure `S` and `combine` function for it. In method `build` we just change `+` to this function. In `modify` we need to ensure the correct ordering of children, knowing that left child has even index. When answering the query, we note that nodes corresponding to the left border are processed from left to right, while the right border moves from right to left. We can express it in the code in the following way:

```
void modify(int p, const S& value) {
    for (t[p += n] = value; p >>= 1; ) t[p] = combine(t[p<<1], t[p<<1|1]);
}

S query(int l, int r) {
    S resl, resr;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l&1) resl = combine(resl, t[l++]);
        if (r&1) resr = combine(t[--r], resr);
    }
    return combine(resl, resr);
}
```

Lazy propagation

Next we'll describe a technique to perform both range queries and range modifications, which is called lazy propagation. First, we need more variables:

```
int h = sizeof(int) * 8 - __builtin_clz(n);
int d[N];
```

h is a height of the tree, the highest significant bit in n . `d[i]` is a delayed operation to be propagated to the children of node i when necessary (this should become clearer from the examples). Array size is only `N` because we don't have to store this information for leaves — they don't have any children. This leads us to a total of $3 * n$ memory use.

Previously we could say that `t[i]` is a value corresponding to its segment. Now it's not entirely true — first we need to apply all the delayed operations on the route from node i to the root of the tree (parents of node i). We assume that `t[i]` already includes `d[i]`, so that route starts not with i but with its direct parent.

Let's get back to our first example with interval $[3, 11]$, but now we want to modify all the elements inside this interval. In order to do that we modify $t[i]$ and $d[i]$ at the nodes 19, 5, 12 and 26. Later if we're asked for a value for example in node 22, we need to propagate modification from node 5 down the tree. Note that our modifications could affect $t[i]$ values up the tree as well: node 19 affects nodes 9, 4, 2 and 1, node 5 affects 2 and 1. Next fact is critical for the complexity of our operations:

Modification on interval $[l, r)$ affects $t[i]$ values only in the parents of border leaves: $l+n$ and $r+n-1$ (except the values that compose the interval itself — the ones accessed in *for* loop).

The proof is simple. When processing the left border, the node we modify in our loop is always the right child of its parent. Then all the previous modifications were made in the subtree of the left child of the same parent. Otherwise we would process the parent instead of both its children. This means current direct parent is also a parent of leaf $l+n$. Similar arguments apply to the right border.

OK, enough words for now, I think it's time to look at concrete examples.

Increment modifications, queries for maximum

This is probably the simplest case. The code below is far from universal and not the most efficient, but it's a good place to start.

```
void apply(int p, int value) {
    t[p] += value;
    if (p < n) d[p] += value;
}

void build(int p) {
    while (p > 1) p >>= 1, t[p] = max(t[p<<1], t[p<<1|1]) + d[p];
}

void push(int p) {
    for (int s = h; s > 0; --s) {
        int i = p >> s;
        if (d[i] != 0) {
            apply(i<<1, d[i]);
            apply(i<<1|1, d[i]);
            d[i] = 0;
        }
    }
}
```